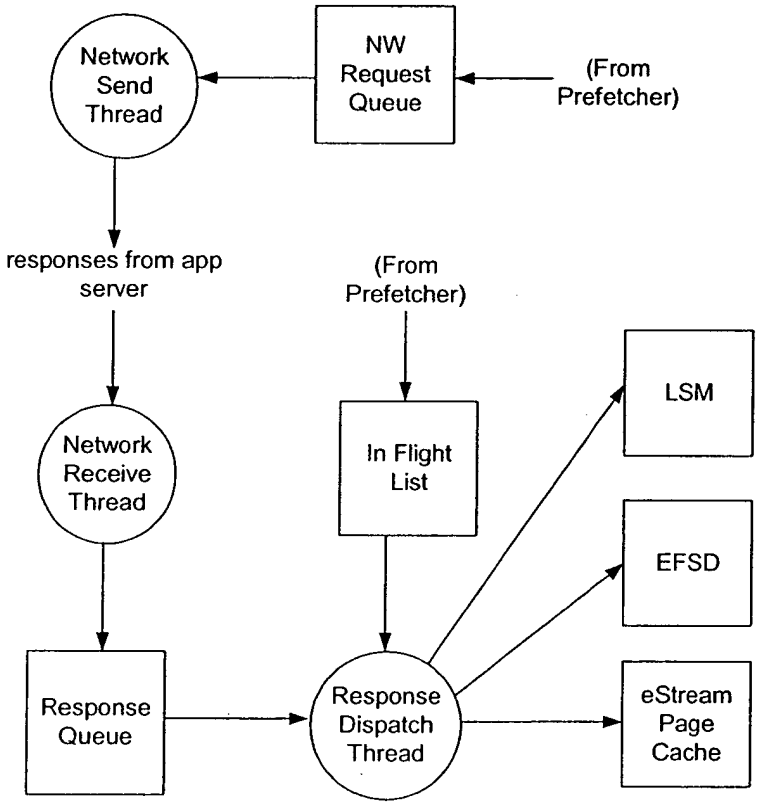
The network send thread is periodically awoken, and it coalesces requests off the NW request queue and sends them to the server. Unlike in the synchronous model, this thread does not synchronously wait for the request to come back from the server. Instead, it simply sends requests until the queue is empty, then goes back to sleep.

The network receive thread waits for responses to come back from any server. Because of the EMS's asynchronous call implementation details, this thread posts returned data to a queue of responses to be handled by another thread. The network receive thread is also responsible for handling timeouts and reissuing those network requests on different servers.

Finally, the response dispatch thread pulls responses off the response queue, and handles the work of dispatching them appropriately.



Handling Network Failure

When the client networking component is notified of a message failure by the EMS, the client worker thread will attempt to reissue the request on a different server.

Coalescing Multiple Requests

The CNI will coalesce multiple page requests that come from the LSM into a single request to an application server. Multiple pages requests for the same application may be coalesced. No other types of requests may be coalesced, including page requests for dif-

ferent applcations. The CNI will not produce requests larger than the maximum allowed by the application server.

Handling Persistent Failures
There will be some persistent failures that will result in the network being unable to fulfill page requests in a timely fashion. This may be due to network or server failure. (These may be indistinguishable from the CNI's point of view.) When the CNI has failed to satisfy a request for a certain amount of time, it will need to ask the user if he wants it to continue retrying, or if it should let the application terminate. It will do this via the **CUIAskUserYesNo()** interface. The client software control panel should include an option to always wait until the server is available, and never ask the user if he wants the application terminated.

Testing design

Unit testing plans

The testing harness for the networking component will be a set of dummy EMS drivers and a dummy NW client. The dummy EMS driver will be capable of performing a variety of actions, including returning appropriate responses, returning inappropriate responses, and timing out without any response. The dummy NW client will have knowledge about the expected EMS behavior, and will verify that the data it gets back from the network component are as expected.

Stress testing plans

Failure testing plans

The client NW is the sole component responsible for implementing server failover. In order to test this code, it is necessary to implement a server with predefined bad behavior. The server failure modes that must be tested include
- server that accepts a connection on a socket but doesn't respond to any requests
- server that closes the socket before sending a response
- server that closes the socket in the middle of a response
- server that sends a partial response and then just stops
- server that satisfies n requests then closes the socket or refuses to service more

It is important that we cover scenarios that look like network failures and ones that look like server failures. (Are there other failure modes that are interesting?)

Cross-component testing plans

Cross-component testing of the client NW includes integration testing with the EMS, the LSM, and the prefetcher. Testing with the EMS can be performed in a manner similar to unit testing in conjunction with a specially written server. Testing with the LSM or pre-

fetcher can be performed in isolation by writing drivers for either the LSM or prefetcher, and using a dummy or real EMS. I'm not sure if this sort of testing is worth the effort to write the appropriate harnesses. Verifying the output of such a combined system is certainly trickier than testing any component in isolation.

Open Issues

Exhibit C3

eStream 1.0 Cache Manager Low Level Design

Version 1.4

[redacted]

Omnishift Technologies, Inc.
Company Confidential

Functionality

The eStream cache manager implements much of the client-side functionality for handling the eStream file system. The cache manager handles all file system requests made by the operating system by reading information from the cache or by passing the requests along to the profiling and prefetching component to fetch missing data from the network.

The cache manager will initially be implemented in user space, but it may be useful to migrate it to the kernel for improved performance. In user space, it will be part of the eStream client process. In the kernel, it will probably be a device driver distinct from the eStream file system driver.

The cache manager manages the on-disk cache of file system data, and the in-memory data structures for managing this cache. It does not manage prefetching of data from the server; that is the role of the eStream Profiling and Fetching (EPF) component. A separate networking component handles the network traffic. This component will also be described separately.

Since there is no overall discussion of the client architecture at a more detailed level than the high level design, this document will cover that as well.

Multiple cache page files will be supported. Each cache page file may be up to 2 GB in size. Different cache files may reside on different or the same logical disk (i.e. Windows drive letter.)

Data type definitions

An application ID uniquely identifies an eStream application. Just what constitutes "one" eStream application is not entirely defined, but different "builds" of the "same" app will be considered different eStream applications. For example, the Chinese-language version of Office is a different eStream application than the English-language version.

```
typedef uint128 ApplicationID;
```

The eStream page number is the data type used to describe a page number within a particular file. Note that this is a page offset, not a byte offset. For eStream 1.0, the cache manager will only support 2 GB cache files.

```
typedef uint32 EStreamPageNumber;
```


[redacted]

The fileId is used to uniquely identify a file within the universe of all eStream files across all eStream applications.

```
typedef struct {
     ApplicationID App,
     int32 File
} fileId;
```

The eStream page size is the fundamental size for eStream requests. This size is in bytes.

```
#define ESTREAM_PAGE_SIZE 4096
```

The eStream file system uses the file time format of the Windows operating system. If the client runs on a system with a different native time format, the client software will be responsible for translating between the native format and the eStream format. The Windows data format is a 64-bit counter of the number of 100-nanosecond periods since January 1, 1601.

EStream metadata is the file information supported by the eStream file system. This metadata is independent of the client or server operating system.

```
typedef struct
{
     uint64 CreationTime;
     uint64 AccessTime;
     uint32 FileSize;
     uint32 FileSystemAttributes;
     uint32 EStreamAttributes;
} Metadata;
```

The eStream inode contains the layout of a file in the cache. Each inode has the following structure:

```
typedef struct
{
     FileId Id; /* ID of this file; search parent for
name*/
     Metadata Metadata;
     FileID Parent; /* parent directory's file id */
     uint32 NumPages;
     PageInfo *Pages;
} EStreamInode;
```

The PageInfo array is variable sized. There is one entry in the pages array for each page in the file (not for each page cached, since we need to know whether the pages are present or not...) Note that the inode is only used in the "robust" implementation.

```
typedef struct
{
     EStreamPageNumber CachePageNumber;
     PageStatus Status;
     unsigned char Priority;
     PageChecksum Checksum;
} PageInfo;
```

The page number doesn't require the 32 bits, since pages are 4096 bytes long. The extra bits will be used to encode which cache file this page resides in. The priority field is a number representing this page's priority for being kicked out of the cache. How exactly this field is used hasn't yet been determined. The checksum is a (fast) page checksum that can be used to validate the contents of this page. Note that it will be useful to have a slower, more effective checksum for development and a faster (but less thorough) checksum for deployment.

The page status is an enumeration for the page's locking status (these are described in more detail later:

```
typedef enum
{
     PS_INVALID,
     PS_CLEAN_UNLOCKED,
     PS_CLEAN_LOCKED,
     PS_DIRTY_UNLOCKED,
     PS_DIRTY_LOCKED,
     PS_IN_FLIGHT
} PageStatus;
```

Note that this describes the layout of the tables in memory; how these data structures are represented on disk is described later.

The EFSD file handle is a small integer passed between the EFSD and the ECM. This is used opaquely by the EFSD and is used as an index into an open file table by the ECM.

```
typedef uint32 EFSDFileHandle;
```

The ECM request type specifies the request type to the rest of the system. Note that some "requests" are used to inform the prefetcher about the events handled solely by the ECM, and do not actually request that any particular action be taken by the prefetcher.

```
typedef enum
{
     ERT_READ,
     ERT_WRITE,
     ERT_READ_HIT,
```


```
    ERT_WRITE_HIT
} ECMRequestType;
```

The ECM request is a request descriptor that is used in various lists within the cache manager. These lists are doubly-linked, circular lists.

```
typedef struct _ECMRequest
{
    uint32 RequestID; /* same as EFSD request id */
    ECMRequestType RequestType;
    union {} Parameters; /* union of all parameters*/
    struct _ECMRequest *next;
    struct _ECMRequest *prev;
} ECMRequest;
```

The cache manager must maintain an array of files that have currently been opened by the EFSD. This array will be statically allocated. This will put a limit on the number of files that may be opened concurrently on the eStream file system. The elements of the array are the following:

```
typedef struct
{
    uint32 Valid;
    fileId File;
    HANDLE OpenFile; /* for simple implementation */
    eStreamInode *Inode; /* for robust implementation */
} OpenFileInfo;
```

The cache manager maintains a hash table containing information about each application that currently has open files. The hash table is indexed by app ID, and contains the following active app information records:

```
typedef struct
{
    AppID App; /* identity of this app */
    uint32 OpenFiles; /* # of open files */
    uint32 HaveAccessToken; /* boolean */
} ActiveAppInfo;
```

The ECM will use this table to quickly determine whether it should continue processing a request it gets from the EFSD, or if the request should be passed to the LSM to ensure that an access token is available. See the section below on ECM-LSM interaction for more details.

The LSM uses the access token state to specify a state for an access token. Right now, we only plan to support valid and invalid, but it may be interesting in the future to allow already opened files to be read, but no new files to be opened.

```
typedef enum
{
      ATS_INVALID,
      ATS_VALID,
      ATS_VALID_NO_OPEN
} AppTokenState;
```

Interface definitions

The ECM exports the following interfaces for operating on the cache. They may be called by the cache manager, prefetcher, or networking component. (Not all components are expected to call all interfaces; see each interface description for more details.)

Note that the cache interfaces are defined at a very high level as the actions that may be performed on the cache by the components, such as enqueuing a new request. They have been defined this way so that these intrinsic operations can be implemented correctly once and limit the possibility that an individual component will not perform proper actions.

ECMReservePage

eStreamStatus ECMReservePage(
IN fileId *File,*
IN EStreamPageNumber *Page,*
IN ECMRequest **Request*
);

ECMReservePage reserves a page in the cache for a request. This interface is called by the prefetching component, and will send a request to the network component. Logically, this interface reserves an empty cache page for this request (if one is available), puts this request on the "in flight" queue, and calls on the network to request the page (unless it is already in flight.)

ECMIsPageInCache

eStreamStatus ECMIsPageInCache(
IN fileId *File,*
IN EStreamPageNumber *Page*
);

ECMIsPageInCache returns TRUE if the specified block is in the cache, and FALSE otherwise. It is used by the EPF to determine if it should prefetch a block; normally, the EPF would choose not to prefetch something that is already in the cache. Note that it would be a good idea for the prefetcher to adjust the priority of a page that it thinks it wants to prefetch, so that they are less likely to be evicted from the cache before they are needed.

ECMDeplanePage

eStreamStatus ECMDeplanePage(

IN fileId *File,*
IN EStreamPageNumber *Page,*
IN char *Buffer[ESTREAM_PAGE_SIZE]*
);

ECMDeplanePage performs all the necessary actions for writing a page coming off the network into the cache and back to the EFSD. This consists of copying the page into the cache, remove all pending requests for this page from the in flight list, marking the page as clean/unlocked, and returning the page to the EFSD for each in flight request.

ECMReadPage

eStreamStatus ECMReadPage(
IN fileId *File,*
IN EStreamPageNumber *Page,*
IN ECMRequest **Request*
);

ECMReadPage performs all the necessary actions for attempting a page read from the cache. The cache is checked to see if it contains the page; if so, the page is copied to the buffer, the EPF is notified of the hit, and appropriate status is returned. Otherwise, this page is put on the queue for requests pending to the prefetching component, and appropriate status is returned.

ECMWritePage

eStreamStatus ECMWritePage(
IN fileId *File,*
IN EStreamPageNumber *Page,*
IN ECMRequest **Request*
);

ECMWritePage performs all the necessary actions for attempting to write a page in the cache. Note that this could be somewhat more complex than a read, because a partial write to a page might necessitate reading the page from the server before writing the partial page to the cache.

The following interfaces are the abstract interfaces that the ECM will use to communicate with the EFSD. Hiding the EFSD's raw DeviceIoControls behind these interfaces will help make porting the ECM into the kernel easier, should we decide to do that.

ECMSetTokenState

eStreamStatus ECMSetTokenState(
IN AppId *App,*
IN AppTokenState *State*
);

ECMSetTokenState is called by the LSM to indicate to the ECM that a token has become available or has expired. The main effect of this interface is to update the state of the specified application in the active app table. See the ECM-LSM interaction below for more details.

ECMGetCacheInfo

eStreamStatus ECMGetCacheInfo(
OUT UNICODE_STRING *Location*,
OUT uint32 **CurrentSize*,
OUT uint32 **MaximumSize*
);

ECMGetCacheInfo is called by the client user interface to find out where the ECM cache is located and its current and maximum size. Location is an absolute path name of the cache file.

ECMSetCacheInfo

eStreamStatus ECMSetCacheInfo(
IN UNICODE_STRING *Location*,
IN uint32 *MaximumSize*
);

ECMSetCacheInfo is called by the user interface when a new cache location or size has been requested. Note that the cache manager may only begin using the new cache information after a restart of the client software (which may only occur on client machine reboot.) The client UI will call this interface when it wants to make a change; the ECM is responsible for actually resizing the cache and making any changes necessary to persistent storage (i.e. the registry).

EFSDGetRequest

eStreamStatus EFSDGetRequest(
OUT EStreamRequest ***Request*
);

EFSDGetRequest reads the next request from the EFSD, including any parameters that need to be passed. This may involve one or more DeviceIoControl calls to the EFSD. **EFSDGetNextRequest** is responsible for allocating memory for this request, and an **EFSDCompleteRequest** call will be repsonsible for deallocating the memory.

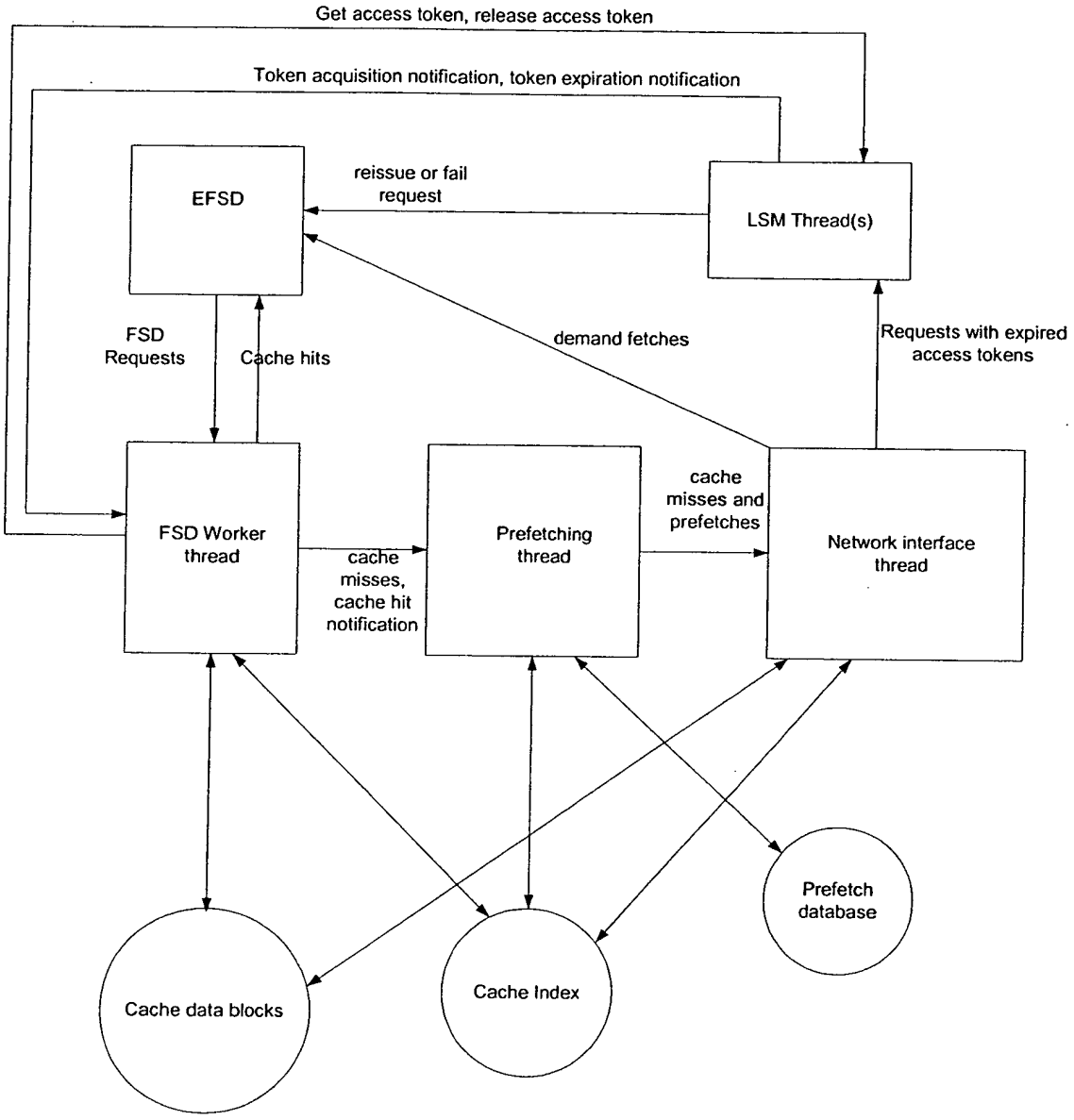
EFSDCompleteRequest

eStreamStatus EFSDCompleteRequest(
IN EStreamRequest **Request*,
IN ECMErrorCode *Status*
);

EFSDCompleteRequest will be called for each request that is received by the ECM via EFSDGetRequest. *status* indicates the completion status for this request, and may indicate success, a retry, or a particular failure condition. Non-persistent errors will be handled by the ECM internally or by requesting a retry of a particular request. Errors reported to the EFSD will be propagated up the file system stack.

Overall Client Architecture

The eStream client will have various types of threads in order to perform its work. The basic architecture is illustrated by the following diagram.



The FSD worker thread will pull requests from the FSD. It will return data for requests that can be satisfied immediately. Any request that requires information that is not currently in the cache will be put on a queue for the prefetching thread to handle.

The profiler will receive all cache misses from the FSD worker thread. Using its own data structures (which may include information about recent cache misses in addition to information about general prefetch patterns), it will decide which blocks it should prefetch. Demand fetch and prefetch requests are sent to the network component. The

only way demand fetches and prefetches are treated differently by the network component is that demand fetches are sent to the EFSD while prefetches are not.

The network thread will manage open connections to app servers and retry requests that time out. When data comes back from the network, the network thread will copy the returned buffer into the cache and to the FSD, if the request was a demand miss.

The cache manager consists of the EFSD worker thread and the APIs to access the cache index, the data blocks, and various queues used by threads in the client.

Not shown on the diagram is an error thread. This thread is responsible for calling the client UI module indicating appropriate error messages and waiting for the user's input. When any component decides that it has an error condition that requires user input, it calls **ECMReportError** with the request and an appropriate error condition, which will be enqueued for the error thread to handle. For example, when the network interface times out reading a page from an application server enough times, it will call **ECMReportError**. When the error thread gets to this request in the queue, it will ask the user if he wants to wait until the app server is available or allow the application to terminate.

ECM-LSM Interaction

The ECM-LSM interaction is a relatively simple one. The LSM notifies the ECM when it first receives an access token and when its access token expires. It does this via the **ECMSetTokenState** interface. The ECM keeps track of each application that has had files open, and whether or not we have an access token for each of these apps.

App ID	# of open files	Have access token?

Note that the LSM need not notify the ECM of mundane events like renewals as long as some token is valid. Also, the ECM does not keep track of the token itself, just whether or not we have a valid one. An additional nicety of this approach is that we could allow the ECM to satisfy requests out of the cache as if we have an access token, without actually having one.

When it receives a request, the ECM checks its table to determine if an access token is available. If it is, it handles the request as normal. If not, it asks the LSM to acquire an access token via **LSMGetAccessToken**. The LSM may return that it has a token, in which case the ECM will continue to process the request, or the LSM may say it doesn't have a token, in which case the LSM takes ownership of the request and will reissue the request when the access token is available.

When the number of open files drops from 1 to 0, the ECM will mark the token as invalid in its table and call **LSMReleaseToken**. The LSM may choose not to renew access tokens that have been released.

Component design

Two cache organizations will be presented. One is suitable for a quick implementation but doesn't lend itself particularly well to high performance or easy manageability; the other will be more difficult to implement but should provide better performance. I will first describe some data structures that are shared by both designs, then go into the specifics of each design.

Common Data Structures and Algorithms

Certain request lists are common to both cache organizations. One is a queue between the FSD worker thread and the prefetching thread for demand fetches that have not yet been seen by the prefetcher. The other is a list of all requests for pages that are "in flight." Requests from the in flight list are removed when they have been satisfied. The in flight list is unsorted and searched whenever a request comes back for requests that match the returned page. If the performance of this data structure becomes an issue, we will change its organization for faster lookup.

Both request lists use the request data structure described above.

The ECM will maintain an array of files currently opened by the EFSD. On file opens, an empty location in this table will be allocated for the newly opened file, and the index to that entry returned as the file handle. (Note that the way the interface between the ECM and the EFSD is defined, it is an error to open an already opened file. The cache manager will have to detect such cases and report an error, but it will not keep a reference count of the number of opens on each file.) This mechanism will allow the ECM to keep track of the volumes that currently have opened files as well as abstracting the client/server file ids away from the kernel driver. (This might allow us to update the client/server protocol without rewriting the EFSD.)

Easier Implementation

The cache will be implemented as a directory tree on the user's hard drive that parallels the eStream file system. Each file will contain a header and an array of status bytes in addition to the data blocks that the file contains. The array of status bytes has one byte for each page in the file. Each byte indicates the current status of that page in the file. (Pages have several different states, so a simple bit per page is not sufficient.) Each file will thus look like

Header
Page Status Bytes
File contents page 0

File contents page 1
...

The header is defined as:

```
typedef struct
{
uint32 magicCookie;
uint32 headerLength; /* Length of this header, in bytes */
fileId fileid; /* for sanity checking */
uint32 length; /* Length of the file, in bytes */
uint32 firstPage; /* Offset to the first page in the file
*/
Metadata metadata;
} ECMCacheFileHeader;
```

The page status bytes begin immediately following the header, and this area is padded with zeros to a page boundary. The first page of the file's contents (and thus each following page of file contents) will therefore begin on a page boundary.

Note that one issue with this design is that files that approach the file size limit of the underlying file system cannot be represented, due to the overhead with the header and bitmap. If this design is used solely for early engineering efforts, then this limitation is acceptable. If we have to work around this limitation, one way to do it is to make the headers and page status bytes reside in a separate file or files.

Directory contents would reside in server format in a file named "Directory" inside of the directory whose contents they represent (with the addition of the header and status bytes as described above for ordinary files). For example, z:\Program Files\Microsoft Office would reside in c:\Cache\Program Files\Microsoft Office\Directory. This has the drawback of creating special file names that can't be used by files in the eStream volume, but again, for an early engineering implementation, this is an acceptable limitation.

Another issue with deploying this implementation is that it is trivial to reverse-engineer this file format and copy files directly from the cache.

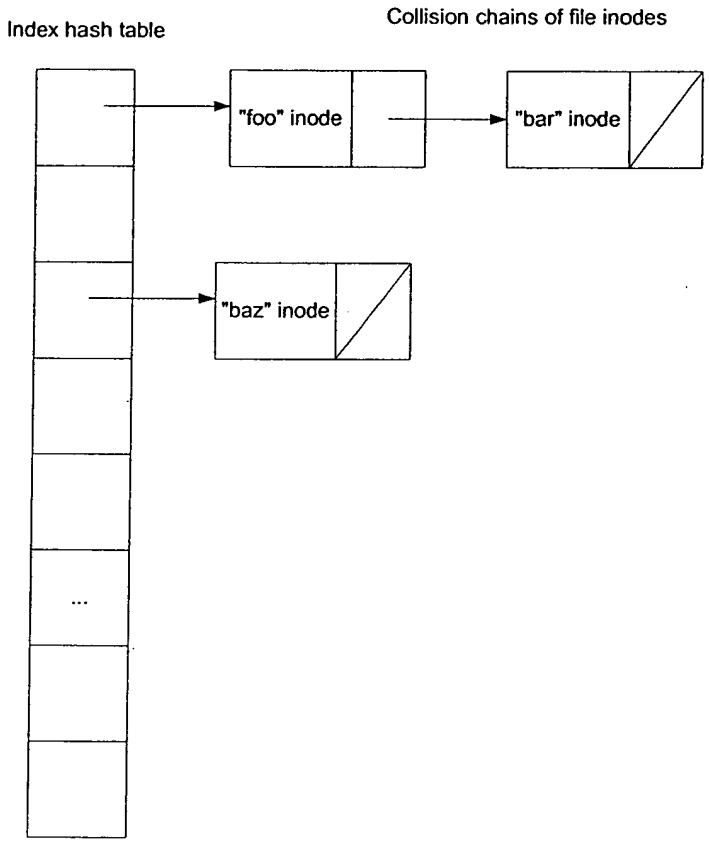
Robust Implementation

The cache will be organized into an index file and one or more cache data files. Multiple data files may be necessary as we may wish to allow the cache to grow larger than the 2 GB file size limit (for some native file systems) or to span multiple drive letters on the client. The data files will only contain pages of file content. These pages will be aligned on page boundaries. The index file contains all the information needed to locate file pages, and is contained in a separate file for simplicity.

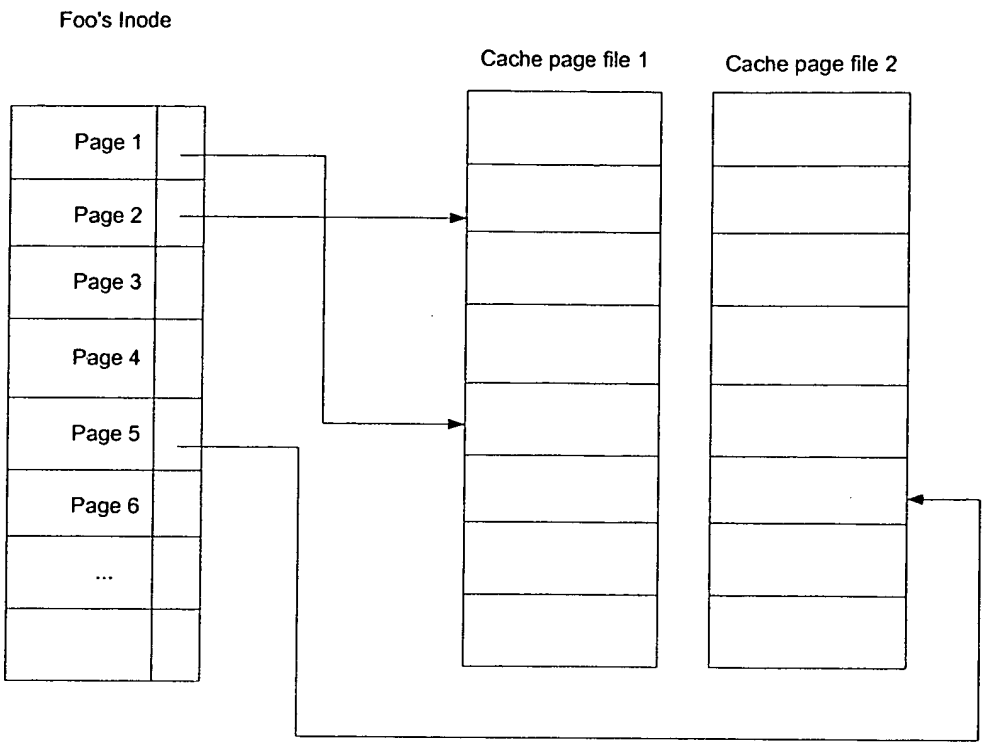
Page and index files must reside on a local disk (rather than a network disk) and cannot be shared by multiple clients.

Each file with any pages currently resident in cache will have a data structure containing information about that file, including its file id, the file id of the directory containing it, the file's metadata, and the map for finding the file's data blocks. This data structure is very similar to the inode of a traditional file system, and will be referred to as the eStream inode. A naive implementation of the inode is described above; no doubt, we will want to reorganize this data structure for more compact representation and better performance. Note that one requirement of the inode is that it contain a status field for each page in the file. One character is sufficient for this status; whether or not we can make do with fewer than 8 bits is an open question.

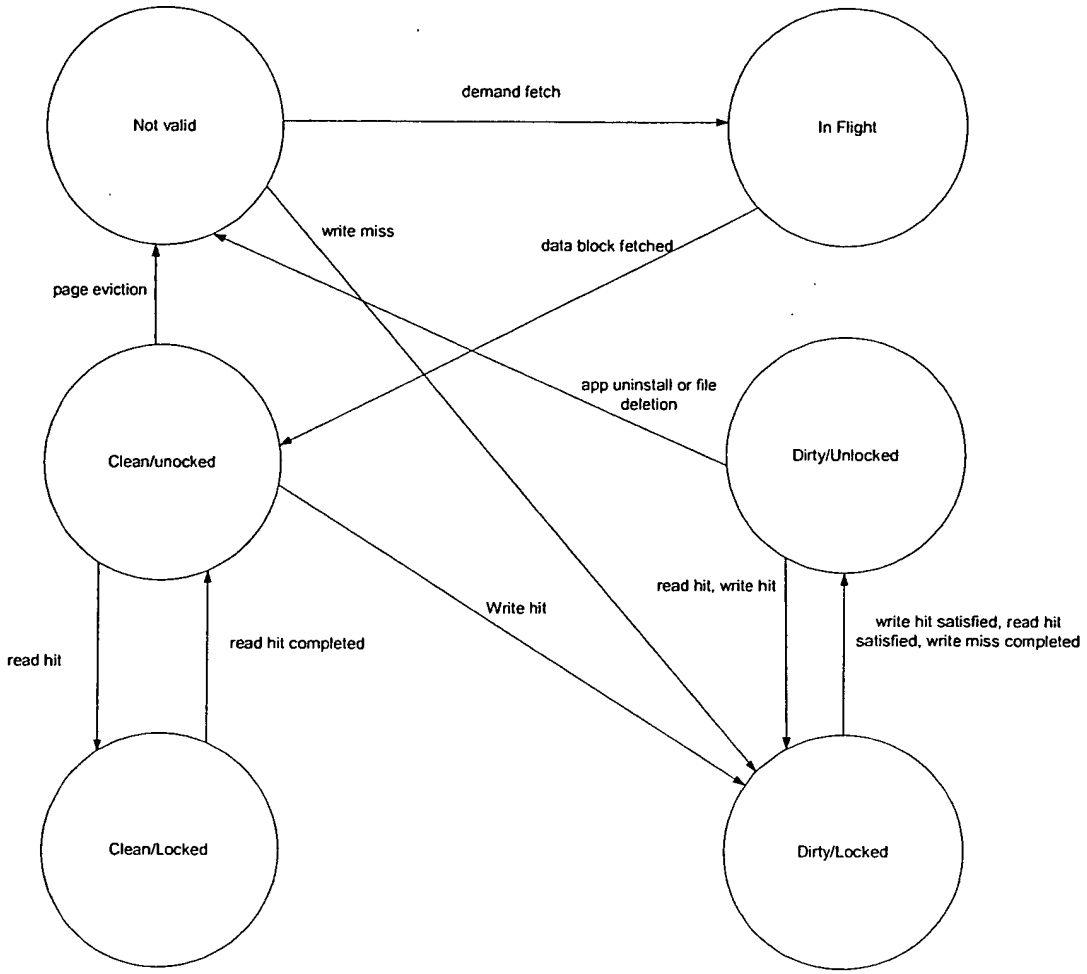
A hash table will be used to map file IDs to file inodes.



The inode contains pointers to each block's location in one or more cache page files:



To prevent race conditions, a single lock controls access to both the hash index and the linked list of requests that are pending network access. Individual pages in the cache may be locked for read or write access. Since each page's status is in the index, the index must be locked order to lock a page for reading or writing. The page states are controlled by the following state machine:



The dirty/clean distinction is between those pages that we have written locally (and thus cannot evict from the cache) and those pages that we haven't written (and thus can be refetched from the server).

A page would be locked while it was being read or written for copying to the file system driver. The operation may thus proceed with the index unlocked, without the possibility of page eviction while a copy is still in progress. The FSD worker thread is the only thread that reads or writes pages from the cache, so it's the only thread that can lock or unlock these pages. The in flight state is only for pages that are currently being fetched, either as a demand fetch or as a prefetch. The prefetching thread is the only thread that will put pages into this state, and only the networking thread will move pages from in flight to unlocked.

A list will be maintained of all "in-flight" requests. A single lock will control access to both this list and the cache index, so there are no race conditions between items being put on this list and data coming off the network. When the FSD worker thread gets a request, it acquires the index lock and looks at the status of the page. If the page is clean or dirty but unlocked, it will lock the page and copy it to the FSD. If the page is invalid, then this is a demand fetch, and the request is forwarded to the prefetcher. If the page is marked in

flight, then this is either a second request for an outstanding demand fetch, or it is a request for an in flight page. Either way, while this thread still holds the index lock, this request will be inserted into the list of in-flight requests. Race conditions might occur because the FSD might make multiple demand reads of the same page, or it may make a demand read to a page that is already in flight due to a prefetch.

Reading requested pages off the network and writing them to the cache (and to the file system driver, if necessary), are where this race condition comes up. We need to ensure that a request for a page that has arrived does not end up in the list of "in flight" requests. The solution is the following: When a data page comes back from the server, the networking component acquires the index lock to find the cache location of this incoming page. If the page is not marked in flight in the cache, this is a bug. (Of course, this is a relatively benign bug, and the NW component could just ignore the page.) The networking thread leaves the page as marked in flight, however, and unlocks the index. It writes the incoming page into the proper location, but it saves the in-memory copy of the page. It then reacquires the index lock, marks the page as clean/unlocked (since it's now in its final location in the cache), removes each request in the in-flight list for this page, then releases the lock. (Any further requests for the same page will find the page clean/unlocked, so the FSD worker thread will be able to satisfy these requests directly.) The networking component then proceeds to satisfy all of the requests it pulled off the in-flight list by using the copy of the page that it saved in memory. This way, it doesn't have to lock the index the entire time it is sending completed requests to the FSD.

Each of these complex scenarios is captured in the cache file's API's. As long as these are implemented correctly, other components don't need to worry about the exact sequence of operations that needs to occur.

Free Space Management

Free pages will be maintained as a free list in memory and as a bitmap on disk. The free list will be built from the bitmap on eStream client software startup. Access to the free list will be controlled by the same lock controlling access to the index.

Evicting Cache Pages

Individual cache pages may be evicted. There is an 8-bit field in the index for each page's importance. Initially, we will implement a random page replacement policy. Later, we will use this page importance field in an unspecified way to replace pages in such a way as to maximize interactive user performance and minimize application server load. Only clean/unlocked pages may be evicted. Pages that are evicted will eventually be put on the free list. Page eviction will only happen at "garbage collection" time. See "crash resilience and garbage collection," below.

Handling Cache Size

Growing the cache should not be an issue. The cache manipulation routines must know the overall size of the cache, in pages. Increasing the size of the cache on the fly should be a relatively straightforward process, as we merely need to lengthen the cache file(s) and add the new pages to the free page list.

Unfortunately, shrinking the cache is a much more difficult operation, since it potentially involves moving around pages that might currently be in use for paging operations or be in flight from the network. Changing the cache around at runtime is both difficult to implement correctly and a performance problem. The current plan is to support shrinking the cache only at eStream client software startup. The maximum allowed size of the cache will be stored in the Registry. On eStream client software startup, the current size of the cache will be compared against the allowed size specified in the registry; if it is larger than the maximum size specified in the registry, then the size of the cache will be reduced by evicting files and compacting the freed space. A request by the user to reduce the size of the cache will take effect the next time the client software starts.

Note that files that the user writes to the z: drive are not considered candidates for eviction (unless the file is explicitly deleted.) This means that the user's on-disk cache may in fact grow to be larger than the limit they specify.

Also note that at least one free page (not used by user-written files) is required for the file system to make forward progress. We also may want to require some minimal amount of cache before eStream will even run. Thus the maximum cache size specified by the user should be considered a "soft limit." There would be a "hard" minimum amount of space equal to the number of pages required to store the files written by the user on the z: drive plus a small amount of cache we designate just for running eStream. If this hard minimum is greater than the soft maximum specified by the user, the hard minimum would win. I would recommend preallocating and non-zero filling the file on disk so that we know that the space is available.

Crash Resilience and Garbage Collection

In order to provide crash resilience, the index will be periodically checkpointed to disk. Note that allocating blocks does not cause problems if the index is not updated. However, we cannot reuse a page's storage until that page has been marked free on disk.

The solution to this problem is to periodically garbage-collect the cache (if it is nearly full), and writing the index to disk. The cache manager will alternate between writing two cache index files. The index file will have a marker at the end that indicates that it has been successfully written and a time stamp, and on startup the ECM will use the latest, fully written index.

Data blocks will always be written directly to the cache page files. These files must be flushed before writing the index.

Garbage collection involves the following steps:
- lock the index
- copy the free list
- choose blocks in the cache to free, and make a list containing just the newly freed blocks. Mark these blocks as invalid in the file's inodes, but don't put them on the free list (yet)

- make a copy of the index
- unlock the index
- merge the list of newly freed blocks with the copy of the free list
- flush all cache page files
- write the new, merged free list (as a bitmap) and index to disk
- lock the index
- add the newly freed blocks to the free list
- unlock the index
- free any allocated data structures

Index File Contents
The index file contains the following items:
- List of cache block files, with their sizes
- Free block bitmap, per cache block file
- Inodes for all files; may be stored hashed or may be rehashed on startup.

Testing design

Unit testing plans

Cache file manipulation routines can be tested in isolation. We will write a standalone harness that exercises the functionality of the cache file manipulation routines by performing cache level operations directly. A multithreaded unit test for the cache manipulation routines would be ideal, so we can test the correctness and performance of our locking strategy without the need to build the entire cache manager.

Each "thread" of execution described by this document can be separately tested by creating a testing harness providing that thread inputs and monitoring its outputs. Replacements for the EFSD interfaces can be very effective here.

Stress testing plans

An interesting stress test for the cache manager is if it can work correctly with very small caches, even all the way down to 1 page. (Or at least, a cache with all pages but one marked as dirty.)

The cache manager will be able to operate in "verify mode," where requests that hit in the cache will still be sent to the server, and the pages returned by the server will be compared with the cached page's contents.

The cache manager will support multiple different page checksum algorithms. We can use a fast algorithm for deployment while using a more rigorous one in development. This also has the benefit of allowing us to test the performance impact of various checksum algorithms.

The cache manager will have the ability to verify the integrity of the cache index and free page bitmap. In particular, it will have the ability to determine taht no pages are allocated to more than one file in the file system, and that each page belongs to a file or is on the free list.

Stress testing for the ECM will include crash testing.

Cache manager testing will include resizing the cache.

Coverage testing plans

Cross-component testing plans

We can build a "cache only" file system by not using the prefetching and network components. This allows us to test the EFSD in conjunction with the cache manager without involving the prefetcher or the network component.

Early implementation of the client will likely involve a null prefetcher that does no prefetching.

We can use the testing harness for the cache manager that doesn't use the EFSD to drive the cache manager in conjunction with the prefetcher and network component. This allows us to test the combination of these components without driving it with the live file system driver.

Upgrading/Supportability/Deployment design

The client user-mode software and device drivers are packaged separately. (I.e. the client executable and the drivers are separate files on the disk.) This leads to the possibility of a "partial" upgrade that results in inconsistent versions of the drivers and client user-mode software. The drivers should support an interface that returns the version number of the driver, or of the interfaces provided by the driver. This will help the client software to recognize situations where it should tell the user to reinstall the client software and not result in bad system behavior.

Most (all?) on-disk data files should have file headers containing at a minimum a magic cookie and the file format version number. This will help us with upgrades in the future.

Open Issues

We need to address what happens when a fetch is requested and no empty space can be found in the cache. The prefetcher should probably block until such time as space is

made available for this request. While operating with very small amounts of cache will obviously cause bad performance, it should not result in a deadlock.

Exhibit C4

eStream Cache Manager Straw Man Proposal

Version 0.2

[redacted]

Purpose

The purpose of this document is to serve as the basis for the design of the eStream Cache Manager. As a straw man, this document is meant to serve as the basis for discussion, and anything here is subject to change. Assuming there are no major concerns with this document, I will proceed with producing a low level design for the cache manager.

Requirements in Brief

Support > 2GB client cache, possibly across multiple drives
Provide some level of protection against piracy, via both the file system and the cache
Fast lookup for what is in the cache and where to find it
Support automatic and user-specified cache size policies

As far as cache size goes, I think that it is reasonable for eStream 1.0 for the cache to be limited to one disk partition and 2GB of space, but the design should allow for very large caches (spanning more than one file and possibly more than one drive letter.) Note that if the cache is greater than 2GB in size, it cannot be mapped into the address space of a single process under NT/2000 on x86.

Cache Organization

The cache will be contained in 2 or more files. One file will contain the cache indices, and one or more files will contain the data blocks for cached files. (More than one cache data file may be required if the cache is larger than the largest file allowed on the native file system.) This allows us to keep the cache index file memory mapped and only map the data file(s) if there is enough memory space to do so.

Data Blocks

The cache data file will contain data pages frome the file system 4k in size.

Data will be stored in the cache uncompressed to allow easy page retrieval.

Cache Index

The cache index will be a b-tree. The key for the lookup will be the file id and page number requested. Keys in the b-tree are the set { volume #, file #, starting page, # of pages }. A lookup will succeed when the volume number and file number match, and the requested page is in the range from starting page to starting page + # of pages. The data stored for that key will be the offset into the cache for the beginning of the run. As is described in the file system proposal, the file number and starting pages are each 32 bits long. I propose making the starting page a 48 bit number and the number of pages a 16 bit number. This allows us to have a very large total cache and reasonable sized runs of contiguous pages in the cache.

Free space in the cache will have to be managed. Free blocks can be placed into a specially identified "free space file" in the index. Some auxiliary data structures may be convenient to make searching for a region of free space of a particular size.

Metadata for a file would be stored in the cache. It would be indexed by page number -1 in the index.

Cache Replacement Policy

For simplicity, I propose that the cache manager evict entire files from the cache when it decides that it needs to clear room in the cache. (Of course, any fragmentary file that is in the cache can be evicted.) We should implement LRU for cache replacement, so we will evict files for apps that have not been run recently.

One Cache Per System

Administrator priviledges are required to install eStream. While various users on a system might have conflicting desires about eStream configuration, such as the size of the cache, I think that it is reasonable to have a policy where the adminstrator controls the setup of the eStream client. By limiting the cache to one per system, we eliminate any ambiguity about cache use in a multiuser environment.

Profiling and Prefetching

Profiling and prefetching have been broken out as a separate component in the client. It will be described elsewhere. It is expected that while the profiler/prefetcher will want access to the cache data structures (i.e. it wants to know what's already in the cache), the logic associated with prefetching is not logically tied to the cache manager, and should thus be separated.

Future Directions

Compression of the cache could potentially be a big win. We could provide cache compression similar to the way that NTFS provides file compression - we compress some number of blocks at a time (e.g. 16) and only store the compressed data when it saves at least one block of storage. Caching of data on disk can sometimes be a performance win, since decompressing the data can be faster than trasferring it on disk if the disk is slow enough.

Exhibit C5

EStream Client Functionality:

- ⇨ Installation of eStream client code
  - o Use browser to contact ASP Web Server, download bits to be installed.
  - o Install z: file system hooks & setup to have z: mounted at system boot.
  - o Install eStream client code, which services z: file sys requests from local cache or from servers & which handles sideband communication w/ servers, and setup to activate estream client code at system boot.
  - o Install NoCluster.sys to disable page fault clustering at system boot.
  - o Install eStream browser plug-in, which can receive messages from ASP Per-User Account Server alerting eStream client when new app purchased. [Sending unsolicited messages may not be possible thru firewall.]
- ⇨ Execution of eStream client code
  - o Respond to z: file sys requests. For apps w/ active online connection(s), user sees the detailed contents on z: that one would see if one had installed the apps locally, though copy access may be controlled. For apps to which the user has obtained offline access, user also sees the appropriate detailed contents on z: (although the files are actually in eStream client-managed memory on local disk). For each app whose connection is currently inactive, user sees a placeholder file entry on the z: file system (on which the user can double-click to launch an active connection).
  - o Establish/terminate session logins to ASP Per-User Account Server, upon user request or upon receiving app purchase message from browser plugin.
  - o Obtain/cache unique certificates for purchased applications from ASP Per-User Account Server.
- ⇨ Register with ASP
  - o Use browser to contact ASP Web Server.
  - o Follow ASP process to register.
  - o User obtains login/password, used by estream client code for sessions.
  - o ASP records user's login/password on ASP Per-User Account Server.
- ⇨ Purchase of application
  - o Use browser to contact ASP Web Server.
  - o Follow ASP process to buy app; user is given unique certificate for app.
  - o App purchase & certificate recorded on ASP Per-User Account Server.
  - o User is directed to go to client & request app installation and/or ASP Per-User Account Server attempts to send message to eStream browser plug-in on user's preferred client system (if any), so client can begin app install.
- ⇨ Installation of application
  - o Send unique certificate for application to appropriate ASP DRM Server, get back id for closest/best App Server & a session id.
  - o Contact designated App Server using id info, download meta-data about app, potentially including registry/DLL/filesys spoofing info, prefetching info, initial cache contents for app. For offline installation, obtain all files.
  - o Perform initial installation & setup for app, after checking system for previously installed version of app & issuing any appropriate warnings.

- ⇨ Execution of application
  - o Send unique certificate for application to appropriate ASP DRM Server, get back id for closest/best App Server & session id.
  - o Contact designated App Server using id info, request file system data as necessary. Respond to running application's requests, collect usage data. Cache portions of application, file system info, & user preference info.
  - o Detect server connection issues (apparent loss of connection or connection response below acceptable threshold); negotiate with ASP DRM Server for alternative connection if need be.
  - o At exit from application (or at other selected times), save portions of cache to client nonvolatile memory. Upload usage information to ASP Per-User Account Server.
- ⇨ Uninstallation of application
  - o Remove all registry/DLL/filesys changes associated with app installation.
  - o Remove all meta-data about app.
- ⇨ Uninstallation of eStream client code
  - o Remove z: file system hooks, eStream client code, & nocluster.sys.

EStream Server Functionality in terms of kinds of eStream Servers responding to Clients [may be embodied in any number of physical server computer systems]:

1. App server
   - o functionally read-only
   - o serves .exes, .dlls, etc.
   - o contains install info (aka, eStream sets)
2. ASP web server
   - o used to get eStream client bits
   - o eStream browser plug-in
   - o handle other user queries, e.g., concerning available apps, current billing status
3. Per-user account server
   - o registration info, issue serial numbers for purchased apps
   - o accept/store uploaded info about app usage
   - o perhaps: user preferences for each app
4. DRM server
   - o authentification of users
   - o validate app licenses, track outstanding offline licenses
   - o hands out licenses for #1 above

Estream Server Management/Maintenance Functionality

- ⇨ Install/maintain eStream apps [aka Builder]
  - o Provide tool/methods to generate initial meta-data about app, including registry/DLL/file spoofing info, initial prefetching info, initial cache contents, etc.
  - o Provide tool/methods to place app & meta-data into public access area and to remove from public access areas
  - o Update meta-data as appropriate to reflect uploaded client usage info
- ⇨ Handle server traffic
  - o Support trouble-shooting of performance or correctness problems

- Perform automated load balancing
- Support online addition/reconfiguration of servers

⇨ Provide tools to process uploaded app usage info.

Open functionality questions:

⇨ Supporting time-based charge for app-usage (e.g., rent by minutes of usage) complicates the design & may engender customer support/satisfaction issues. Do ASPs want/need this support? [Prefer to steer them away from this model.]

⇨ How should we handle minor upgrades/patches of apps (i.e., service packs)? One method to allow active use of previous versions plus availability of new versions without treating new versions as if they were entirely new applications would be file versioning.

Exhibit C6

EStream 1.0 Top Level Component Breakdown * Revision 0.1 [redacted]

Client system components

- Z: File system manager [1]
  - Handles all z: file system requests generated on client
  - Makes requests to EStream cache manager
  - Attempts to filter references that suggest software piracy activity
- EStream client core
  - Session manager [12]
    - Handles establishing/terminating ASP sessions
    - Negotiates for app license & security using user unique certificate
    - Invoked either by eStream client user interface or by cache mgr
  - Cache manager [2]
    - Responds to Z: file system manager requests
    - Maintains client cache of app & file system data/metadata
    - Requests info as necessary from Estream client networking
    - Requests session/license for non-mounted apps from session mgr
    - Consumes/gathers profiling/feedback data
  - File manager [3]
    - Provides interface to all eStream created/maintained client files
    - Gets requests from cache mgr, session mgr, file mgr/spoofer, registry mgr/spoofer, app install/deinstall, client install/deinstall
- Estream client network interface [8]
  - Handles requests from EStream cache manager
  - Handles protocol interface to/from server
  - Performs compression/decompression, encryption/decryption of packets
  - Detects network problems & reports to session manager for renegotiation
- EStream client user interface [5]
  - Displays error/info messages from any part of eStream code to user
  - Solicits/obtains info (e.g., login/password, app license) from user
- EStream file system manager/spoofer [6]
  - Filters all non-z: file sys requests, redirects non-z: file refs as appropriate
  - Supports operation of eStreamed apps
  - Avoids eStreamed apps interfering with non-eStreamed apps
- Estream registry manager/spoofer [7]
  - Filters all registry refs, handles registry contents for/about eStreamed apps
- EStream application installer/deinstaller [14]
  - Obtains app spoofing/registry/prefs info & initial cache/profile data
  - Prepares system to be able to run app on user request
  - Supports deinstallation of app
- EStream client code installer/deinstaller [13]
  - Installs all client Estream code components
  - Supports deinstallation of all eStream components
- NoCluster.sys [4]
  - Disables page fault clustering in the kernel
- Estream browser plugin
  - Optional EStream component which fields unsolicited server messages

Exhibit C7

eStream Client-Server Diagram

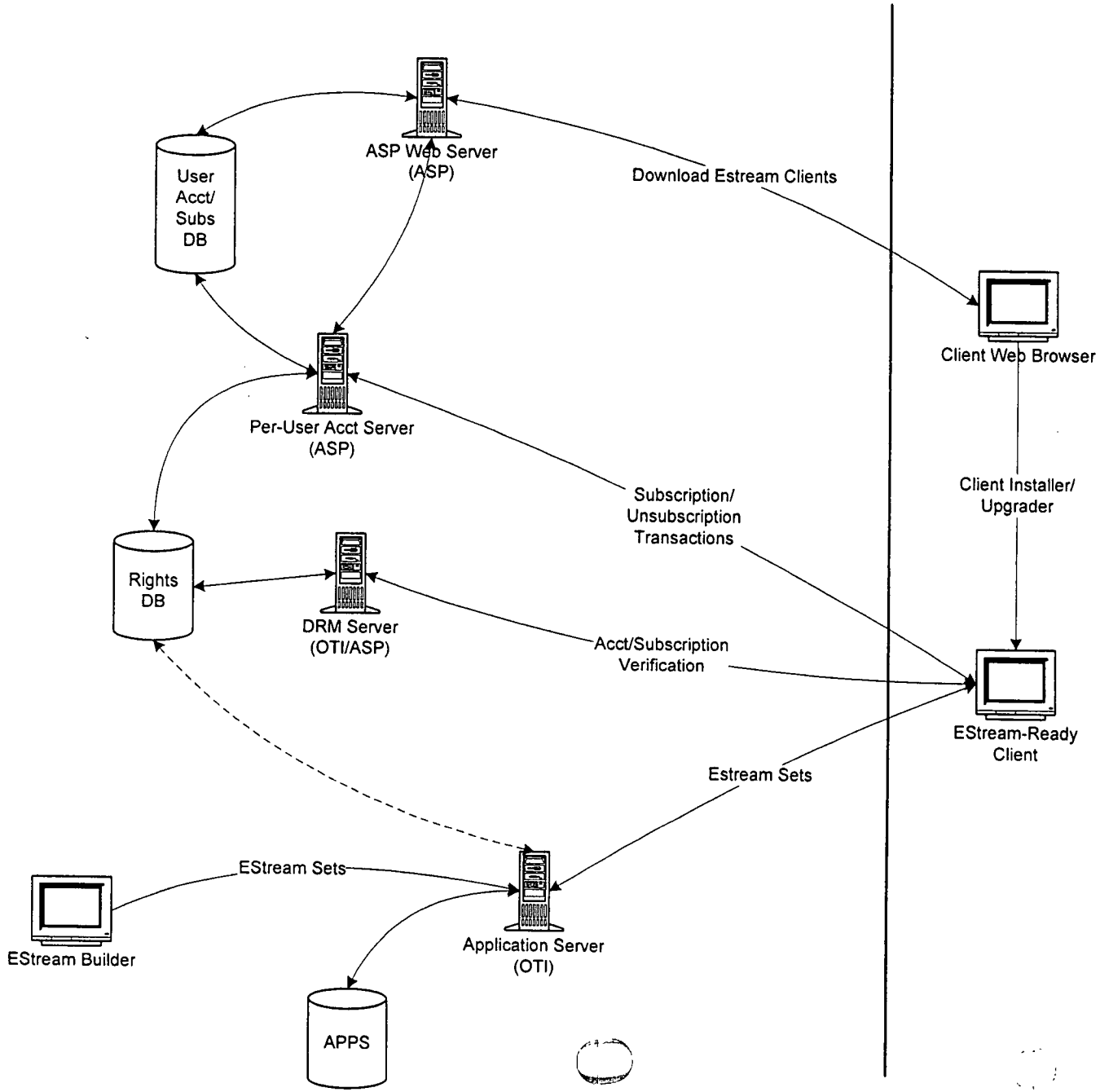


Exhibit C8

eStream File System Straw Man Proposal

Version 0.5

[redacted]

Purpose

The purpose of this document is to present a concrete proposal for the functioning of the eStream file system. In many places, I make some sweeping generalizations about how things should work without describing the data structures and interfaces involved in implementing them. This document should eventually involve into a design specification.

Issues Not Covered

This document does not attempt to cover all issues present in designing the eStream 1.0 product. In particular, the overall authentication/licensing/security architecture is not covered in detail here. It is expected that the security functionality will be mostly orthogonal to the design of the basic file system functionality.

Background

There are a number of different networked file systems out there. Many of them share some requirements with eStream. For example, AFS performs client-side on-disk caching, while Coda handles serious server redundancy and disconnected operation. Personally, I believe that AFS and Coda are the file systems whose designs are most relevant to us. For those interested in further background reading, you might also want to look at papers covering NFS, CIFS, xFS, DFS, and Zebra.

Single File System Name Space

Many modern distributed file systems present the network file system as a single tree mounted at some location on the client system, regardless of which server hosts the data. (In fact, with AFS, every file on every server in the world can be accessed through a path starting with /afs on the client, assuming the client can reach that server and has sufficient privileges to do so.) Compared with systems like NFS and Windows sharing, where each share is mounted in a different location on the client, the single name space provides greater ease of use.

The eStream file system would present one universal logical file system. Regardless of which ASP provider supplies a particular volume, that volume will always be referenced via the same path on the eStream file system. That this is desirable or even feasible is predicated on the assumption that OTI is the only entity providing all eStream sets. Each volume must get a unique identifier and a unique location to be mounted in the file system hierarchy. If two different ASPs privide the same volume ID, then the contents of those volumes must be identical. This way, we don't have to tag things in the cache based on what ASP they came from, and the cache manager doesn't need to know anything about ASPs. If done correctly, only the client networking component and the LSM need to know about ASPs.

Volumes

A volume is a complete subtree of a file system. Volumes may contain files and directories. Volumes may not be mounted in other volumes. A volume is a logical grouping of files within the file system and is the unit of replication across servers. An application will reside in a single volume. Two applications will never share a volume.

Volumes are uniquely identified by a 32-bit volume identifier. Each volume additionally has an 8-bit version number. This version number is incremented each time any file within the volume changes. (See supporting upgrades, below). Note that the volume id is globally unique. If two ASPs provide volumes with the same volume number (and version), they have identical contents.

A volume may be replicated on any number of servers. Each SLM server contains a map describing the application servers that currently provide each volume. This global replication of this table is acceptable because volumes are added or moved infrequently.

Identifying Files

Files and directories are uniquely identified by the pair (volume id, file number). This tuple is called a file id. Volume id and file number are each 32-bit signed integers. Negative values for both volume id and file number are reserved for special purposes, leaving us with 2^31 possible volume IDs and 2^31 possible files per volume.

Finding an Application Server for a Volume

The SLM will tell the client which application servers currently provide each volume. It may be necessary for the client to periodically poll the SLM to get up-to-date information about the state of the application servers. The License and Subscription Manager on the client will keep track of the currently subscribed applications and the application servers for each of these applications.

Directories

Directories are specially formatted files that are used in a special way by the file system. They are identified by file ids, just like other files. From a client-server point of view, they are read by the client in the same way as other files. Directories contain arrays of entries with the following format:
(volume number, file number, flags, length, filename)

The volume number and file number are 32-bit signed integers. The flags are 32-bits of flags. The length is 16 bits and is the length of the filename in bytes. The filename is a non-NUL terminated Unicode string. The structure is padded with enough Unicode NUL characters to make the structure a multiple of 32 bits long. The next directory entry begins on the next 32-bit boundary.

The access token is not part of the directory, as a single access token is required to access all files in a particular volume.

The volume number is required so that the the client can construct a local directory for the root of the directory structure in the same format as other directories (see filename parsing below). It also helps to provide a sanity check.

Accessing Files

Assuming that a client has a file-id for a file that it wishes to access, the following client-server actions must be supported:

For stat-like information on the file, we need a GetFileMetadata() interface. The client would provide the file id it is interested in and the proper access token for this file. The server will either return the metadata for the file or an error condition (like access token expired or incorrect access token.) The metadata contains the standard Windows metadata information, including file length and file access times.

On a file open (CreateFile in Windows terminology), we need to verify that we have access to the requested file. This is probably best accomplished by calling GetFileMetadata and verifying that we can get the metadata. This way, we can fail file opens gracefully if we don't have an access token.

On reads (and writes, when we support them), the client will send the file id and the access token to the server along with an offset and a length for the read and write. The server will respond with the data. Note that the same mechanism will be used for reading both files and directories.

Pseudodirectories

For those parts of the eStream file system name space that do not belong to any volume (such as the root of the file system), the client must construct appropriate directories based on the currently installed applications. This is to support filename parsing starting at the root of the directory. For example, if the client has word installed with a root of /Worddir and it is volume number 3 and Photoshop installed with a root of /Photodir and is volume number 4, the client would construct a directory for the root of the entire file system containing
File name, Volume number, file number
"Worddir", 3, 0
"Photodir", 4, 0

(The file numbers are both zero here because 0 is the index of the root directory of each volume, and these are the mount points for each volume.)

When new applications are installed, the root of the file system would have to be updated to reflect the newly installed apps.

Filename Parsing

Filename parsing is handled one element at a time, starting at the root of the file system. Parsing one path name element involves reading the parent directory's contents (from the

cache or the app server), searching it for the file matching the next path element's name, and getting the appropriate file id so it can do further lookup.

Volume Versioning... Without File Versioning

We can provide volume versioning and incremental volume updates without versioning each file in the file system. When a new volume is to be provided, we can append any new or changed files as new files in the volume, with new volume IDs that weren't already present. If a directory's contents have changed, then a new version of this directory will be built, with a new file number. This process will proceed from the leaves all the way to the root of the file system, eventually resulting in a new root. The old versions of things would still be available for old clients to access, but clients wishing to access the new version will simply start at the new root, and would thereby get to a consistent picture of the volume. Any file or directory that has not changed from the old version to the new one need not be replicated, and will be referenced by its old file number. (I.e. newly reconstructed directories will contain the old file number for any files that haven't changed.)

If we reserve the first 256 file ids for the root directory, then the version number can be the same as the file number for the root directory.

Note that if we decide that the complexity of this approach is too high, this does not preclude always creating a new volume from scratch for each update.

Constructing File IDs

It is the job of the builder to produce the volume file to file id mapping and to construct all of the directories. Because directories are files identified by file id, this process must begin at the leaves of the volume and proceed to the root.

Note that constructing a new changed volume will consist of finding the diffs between the two volumes and producing some new directories. Changed or newly added files will get new file numbers, leaving the old ones around. Note that any directory that has had any descendents changed must be reconstructed with the new file numbers, and the new directory will get a new file number. This process will proceed to the root of the volume, which will receive a new file number.

Server Failover

All app servers for a particular volume must share the same mapping of file ids to file, so server failover is trivial. There might be a performance impact if the new app server doesn't have the requrested file in memory.

Writing Files into the Application Install Directories

Two approaches have been discussed for the problem of applications that want to write files to their install directories. First, this can be handled wholly inside of the eStream file system. The cache manager could allow writes to files handled by the eFS, but these writes would not be written back to the server. Instead, they would simply be written to

the eFS cache and marked non-purgeable. This approach's primary advantage is that it does not rely on a file spoofer.

The other approach is to use the file spoofer to spoof some accesses to the z: drive. Any open for read/write access would cause the existing file (if any) to be copied to a location on the c: drive, and the file spoofer would then redirect the open to the newly created file. The file spoofer would have to keep track of any file created via this copy-on-write mechanism and redirect all future accesses to the copy. There are some issues to this approach. For example, it is extremely wasteful when files on the z: drive are opened for read/write access but are never actually written. However, it does help reduce the complexity of the eFS cache, and is trivial to implement if we have to do c: to z: file spoofing anyway.

In either case, to support the creation of new files in an application's install directory, it must be possible to modify the contents of directories in the cache.

If we don't use the file spoofing approach, there is the issue of how we support written files when we move to a newer version of a volume. It would probably be necessary to walk the cache and make sure that each written file gets placed in the appropriate place in the new volume version. This is likely to be non-trivial, because we need to have full information about the location of each modified file in the file system tree, and would need to download enough of the new volume directory structure to place these modified files there.

64-Bit File Access?
One question we should answer is whether we will support file sizes greater than 2 GB on the eStream file system. I'm inclined to say that such support isn't a requirement for the 1.0 product, but I also think that the implementation and verification complexity of 64-bit file access on the file system is low enough that we might want to consider building it in anyway.

Simplifications
We could preclude the possibility of an application consisting of more than one volume.

Future Possibilities
Epicon seems to make a big selling point of their technology involving "self-healing" of damaged application files. Such support could be provided by computing checksums on files in the cache. Whether or not we want to support this is an open question. My feeling is that it's something we should leave out of 1.0.

Outstanding Issues
Cache organization has not been addressed.

Finding and downloading the app install block has not been addressed.

Security in a multiuser system has not been addressed.

Exhibit C9

A Method for Efficiently and Securely Delivering Computer Applications over a Network

Asignee:
Omnishift Technologies, Inc.
2480 North First Street, Suite 150
San Jose, CA 95131
(408) 321-6900

Manuel E. Benitez
10245 Parkwood Drive # 6
Cupertino, CA 95014
(408) 255-8731

Managing a networked computing environment is a daunting task. The laborious process of ensuring that each computer contains a current version of each application is very time consuming. Several solutions exist to help *Information Technology* (IT) departments reduce application management costs and improve the likelihood that each computer has the appropriate version of each application. These solutions fall into three categories: server-based computing, automatic distribution and application servers.

Server-based computing solutions simplify application manageability by running applications on a *farm* of servers along with mechanisms that deliver the output of the application to a *client* machine and send the keyboard and mouse input back to the server. In this manner, server-based solutions give the appearance of providing the appropriate version of each application on every machine. Using a server-based solution, IT departments reduce application management efforts to managing a server farm. Additionally, applications with modest graphical requirements can be delivered across limited bandwidth network connections and be available during business travel or for telecommuting outside the corporate network. The drawback of server-based solutions is the server farm must provide sufficient computational resources to run all the applications requested simultaneously. Doing so, especially during peak demand periods, requires very substantial investments in servers. Providers of server-based solutions claim that server costs are offset by reductions in the computational requirements placed on client machines. In practice, server-based solutions rarely result in the purchase of cheaper, less-powerful clients because users prefer to retain the freedom to use applications not available through the server-based infrastructure.

Automatic distribution solutions address application availability and versioning issues by providing a mechanism whereby client machines can automatically download new and updated applications from a central server. Automatic distribution solutions consist of a mechanism that takes an inventory of the applications on a client machine and compares it against the current application list. When an update is required, automatic distribution solutions leverage the standard application installation and upgrade processes.

Unfortunately, this requires transferring the entire application to the client, a process that can take minutes across a fast network and hours for business travelers or telecommuters. On the positive side, automatic distribution solutions scale more easily than server-based solutions, as a single server can handle many times as many users and applications.

Application server solutions address application availability and versioning issues by placing all applications on a central storage location. Client computers access these applications through a *network file system* that acts like a standard local file system on a client machine, but in fact provides files stored elsewhere on the network. Many clients can access the same copy of the application stored on a network file system and the IT department can easily upgrade or install applications that can then be used on all client machines. The network file solution works well in cases where network bandwidth is high (10Mb/sec or greater) and latency is low. Current solutions, such as NFS developer by Sun Microsystems, also lack robust security features and are intended for use inside secure corporate LANs.

Background

Figure 1 illustrates a basic computer system. The *central processing unit* executes application instructions that request it to perform operations such as addition, subtraction, multiplication, division and moving data between the various system components. The

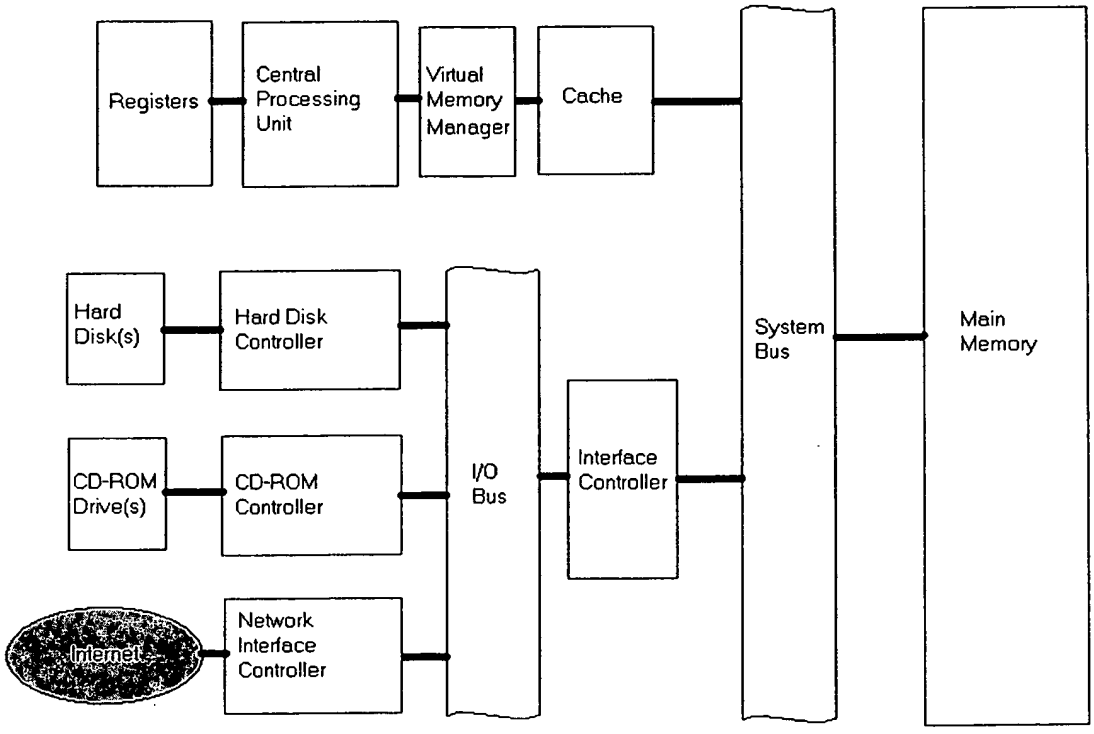


Figure 1: Computer System

central processing unit has two main areas in which it manipulates data: *registers* and *main memory*. Registers are fast but few in numbers. Accessing the main memory takes much longer, but there is much more space to hold data in main memory than there is in the registers. While the central processing unit communicates with the registers directly, its link to main memory and the rest of the is through a communication pipe called the system bus. The system bus coordinates data transfers between system components and operates more slowly than the central processing unit does. Between the central processing unit and the system bus are two special components known as the *virtual memory manager* (whose purpose will be explained presently) and the *cache*. The purpose of the cache is to keep a copy of the most recently referenced data stored in main memory. This is done so that the system bus does not need to be used if these data are referenced again. The cache improves the performance of the system because of a phenomenon called *locality*. Locality dictates that the most recently referenced data are the most likely to be referenced again.

One of the components directly connected to the system bus is the *interface controller*. The interface controller acts as a buffer between the system bus and the slower and more complicated *Input/Output (I/O) bus*. The job of the interface controller is to convey I/O requests from the central processing unit to the *I/O device controllers* and transfer data between I/O device controllers and main memory. Applications running on the computer system are not permitted to communicate with either the interface controller or the I/O device controllers directly. This is because controllers are very sensitive to the timing of events and can easily be put into states where they stop operating properly or start misbehaving so that other components can no longer perform their tasks. The computer system is managed by a special application known as the *operating system*. The operating system is made up of many components. Among these components are a group known as the *device drivers*. The main purpose of a device driver to hide the intricacies of dealing with the I/O interfaces from the rest of the operating system.

I/O device controllers perform the task of controlling the physical or electronic components that make up a device. For example, the *hard disk controller* converts a command to read a particular block of data (called a *sector*) from the hard disk into appropriate levels of electrical current to move the disk's read/write heads to the precise area of the disk in which the sector is located. It also converts the electrical impulses returned by the head's amplifiers into streams of one's and zero's and scans them to determine when the appropriate sector is being read by the head. Once the interface has read the sector and verified through the use of *error detection codes* that it was read correctly, it communicates over the I/O bus to the interface controller and informs it that the sector is ready to be transferred so that it can begin its journey to main memory and, ultimately, the central processing unit.

The *network interface controller* is another component that is commonly found on the I/O bus. Like other device controllers, the network interface controller performs the task of converting commands to send or receive information across an external network connection into the appropriate electrical currents and voltages required to exchange data across the particular type of network that the interface is connected to. The network interface controller also works in conjunction with an appropriate device driver through which applications send and receive data across the network. Because a large part of networking is based on following complex protocols, naming schemes and software-controlled virtual connections, more sections of the operating system are usually located between the network device driver and applications which handle all of the intricacies of splitting long streams of data into shorter ones (called *packets*) which are acceptable to the network. The collection of physical and software components connected together to support network communications is known as the network stack and an instance of one is shown in figure 2. In this illustration, a network interface controller capable of communicating on an Ethernet network is physically connected to an I/O bus using the Peripheral Component Interconnect (PCI) standard. These two components make up the physical or *Hardware* portion of the network stack. The operating system provides a device driver for each of these physical components that are shown at the bottom of the *Software* portion of the network stack. The operating system also provides a component that communicates via the device drivers to implement the Internet Protocol (IP) while another component will use the IP component to implement the Transmission Control Protocol (TCP) and an application might use the TCP component to implement the HyperText Transfer Protocol (HTTP) to implement a web browser.

Having explained the basics of computer systems and network stacks, we turn our

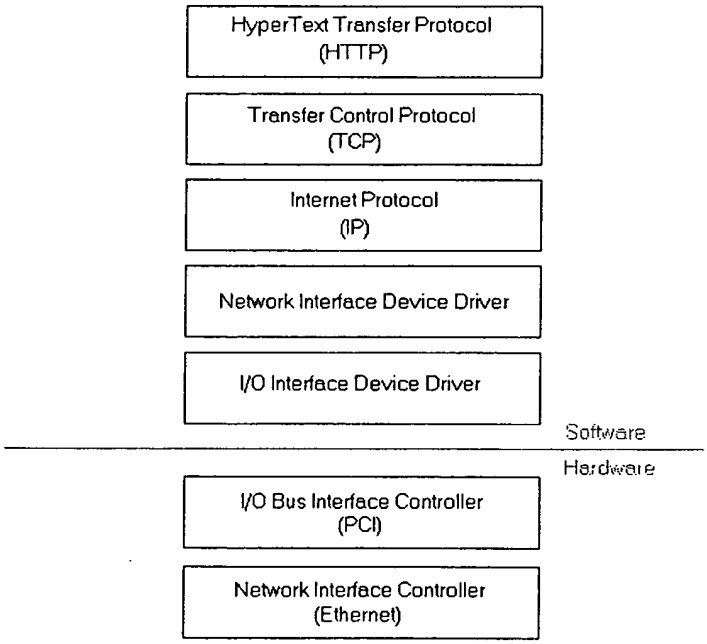


Figure 2: The Network Stack

attention to the virtual memory manager. The central processing unit can reference more main memory than is physically available. The virtual memory manager's task is to mitigate this problem by using a storage device such as a hard disk as an overflow area for data that are not frequently used. This is analogous to keeping documents that are infrequently needed in boxes in the garage. When such a document is needed, its box is brought in from the garage and another box inside the house is selected and taken out to the garage to keep the house tidy. Likewise, the virtual memory system keeps boxes of data called *pages* along with a data structure called a *page table* that keeps track of were each page is currently stored. When the central processing unit needs a piece of data, it sends the *address* of the data to the virtual memory system which quickly determines which page that address is in and consults the page table to determine where that page is located. If the page is in main memory, then the address is modified to location in main memory where the page data actually resides. If the page is not in memory, the virtual memory manager *interrupts* the central processing unit to run another component of the operating system known as the *page fault handler* to obtain the page from the hard disk and copy it to main memory where it can be accessed by the central processing unit. The page fault handler calls on the appropriate device drivers to read the page from the hard disk and copy it to main memory. Before doing this, the page fault handler will usually need to make room for the desired page by selecting another page currently in main memory and writing it to the hard disk so that it can be read back if it is ever needed again.

One of the most fundamental concepts underlying the computer system described here is that *applications* are *data*. They can be wholly or partly stored in main memory or on a hard disk or a CD-ROM disc or, for that matter, on another computer system accessible through the network. Because the central processing unit accesses and manages data in main memory, any part of an application that the central processing unit wants to execute must be brought into main memory. Since there is no difference between applications and data, the virtual memory manager and the cache will handle application components as if they were any other pages of data. One advantage of this is that it allows a computer system to execute an application without having it completely in main memory. The operating system can *map* the application to a set of addresses through the expedient of changing the page table to indicate that the pages comprising the application currently reside on the sectors of the hard disk where the application is stored. When the central processing unit attempts to execute some portion of the application for the first time, the virtual memory manager will interrupt the central processing unit and the page fault handler will copy a page's worth of the application to main memory. The central processing unit can then execute the application instructions in that page without being interrupted until the application strays into a page that has not yet been brought to main memory.

There are numerous advantages to mapping applications via the page table (aka *memory mapping*). The first is that large portions of most applications are never needed except in rare circumstances. For example, very few *spreadsheet* application users make use of *pivot tables*, although a large number of instructions exist solely for the purpose of implementing the pivot table functionality of the spreadsheet application. Bringing all

these instructions into main memory before they are needed would almost always be wasted time and effort. In fact, only a small portion (about 10-20%) of most applications is referenced most (80-90%) of the time. This phenomenon was introduced earlier as *locality*. Memory mapping thus improves performance by reducing the amount of application code that need to be transferred from the hard disk to main memory. Also, since only 10-20% of an application is needed at any given time, memory mapping allows an operating system to simultaneously run several applications while using only the amount of main memory that would be needed to hold one entire application.

Technical Description of the Problem

Locality and memory mapping would then seem like the perfect solution to executing applications across a network. Rather than having the application installed and stored on a computer system's hard disk, it can be kept somewhere on the network and paged in as needed. There are several problems with this approach, however, namely: bandwidth, latency and security.

To handle a request for a page from a hard disk, the page fault handler would determine from the page table which sector on which disk the page was located in. The following steps need to then take place:

1. The read sector command is sent to the disk device driver which will place it on a queue where it will wait until all previous commands for that disk have been sent and the hard disk controller indicates that it is ready to receive its next command
2. The read sector command and the address of the appropriate hard disk controller are passed to the interface controller device driver, which places them on a queue where it will wait until all previous commands for the interface controller have been sent and the interface controller indicates that it is ready to receive its next command
3. The interface controller waits for the I/O bus to become available and transmits the read sector command to the hard disk controller
4. The hard disk controller determines where it needs to position the read/write head and sends appropriate levels of current to the head controller to move the head
5. The head controller and head are physical devices which obey the laws of physics and must accelerate, cross the distance in space towards where the sector is located and then decelerate to stop the head at the appropriate location
6. The disk platter, which is also a physical device, is spinning at a constant number of revolutions per second and the disk controller must wait until the desired sector begins to travel under the head so that the sector can be read
7. The hard disk controller reads each bit of each byte that makes up the sector and its error detection codes and stores them in a small memory buffer, the rate of speed at which this happens is determined by how long it takes the spinning platter to rotate across the length of the sector
8. The sector is then verified by the hard disk controller by examining the sector data and the error detection codes

9. The hard disk controller waits for the I/O bus to become available and transmits a message to the interface controller informing it that it has the requested sector
10. The interface controller waits for the I/O bus to become available and transmits a message to the hard disk controller requesting that it send the sector data over the I/O bus
11. The hard disk controller waits for the I/O bus to become available and transmits the sector data to the interface controller
12. The interface controller places the sector data in a memory buffer
13. The interface controller waits until the system bus is available and then transfers the sector data to main memory
14. The interface controller sends a request on the system bus to the central processing unit indicating that it would like to communicate with its device driver
15. The central processing unit places the request on a queue and, when it is ready, begins to execute the device driver
16. The device driver determines which command has successfully completed, removes it from its queue and informs the operating system that its should execute the disk drive device driver because something that interests it has happened
17. The hard disk device driver determines which command has successfully completed, removes it from its queue and informs the page fault handler that its page is now located in main memory
18. The page fault handler updates the page table to reflect the new location of the page and informs the central processing unit that it can resume executing the application that caused the page fault

This represents a substantial amount of work. Fortunately, most of these operations are completed very quickly given the tremendous computational capacity of a computer. The most time-consuming items are the ones that transpire in the physical domain. Moving the disk head takes about 8-12 msec and waiting for the platter to rotate another 0.2-0.5 msec. Another significant factor is the time spent transferring the page over the I/O bus whose bandwidth is in the 20-80 Mb/sec range. For standard 4 KB pages, this consumes between 0.4 and 1.6 msec of time. Translating this into real time as might be perceived by a human user, if a large application incurs 1000 pages faults (4 MB, average for a 20-40MB application) the system would spend about 12 seconds handling page faults. This would be spread out across the execution of the application with roughly one-third of it happening when the application is initially started. Since large applications usually execute for many minutes, the overall time spent handling page faults tends to be unnoticeable to a user except at the very start of the application or when the system is pushed beyond the point at which the physical memory available can hold the portions of the applications that it is executing. The latter situation is known as *thrashing*, which is characterized by constant disk activity and very little useful progress.

Suppose that the application were to reside on another computer system and the virtual memory manager could access this via the network interface controller. The previous page fault scenario would now be handled as such:

1. The read page command is sent to the appropriate layer of the network stack (most likely the HTTP layer) and would work its way to the network interface device driver which will place it on a queue where it will wait until all previous commands for that network interface have been sent and the network interface controller indicates that it is ready to receive its next command
2. The read page command and the address of the appropriate network interface controller are passed to the interface controller device driver, which places them on a queue where it will wait until all previous commands for the interface controller have been sent and the interface controller indicates that it is ready to receive its next command
3. The interface controller waits for the I/O bus to become available and transmits the read page command to the network interface controller
4. The network interface controller waits for the network to become available and sends appropriate levels of current across the network to send the request for the page
5. The message is received by the computer system that contains the page
6. The page is placed on the network
7. The network interface controller detects the reply and reads each bit of each byte that makes up the page and its error detection codes and stores them in a small memory buffer, the rate of speed at which this happens is determined by the bandwidth of the network
8. The page may have been broken up into a number of smaller packets, in which case the network interface controller waits for each packet to arrive and reconstructs the original page in a memory buffer
9. The page is then verified by the network interface controller by examining the page data and the error detection codes
10. The network interface controller waits for the I/O bus to become available and transmits a message to the interface controller informing it that it has the requested sector
11. The interface controller waits for the I/O bus to become available and transmits a message to the network interface controller requesting that it send the page data over the I/O bus
12. The network interface controller waits for the I/O bus to become available and transmits the page data to the interface controller
13. The interface controller places the page data in a memory buffer
14. The interface controller waits until the system bus is available and then transfers the sector data to main memory
15. The interface controller sends a request on the system bus to the central processing unit indicating that it would like to communicate with its device driver
16. The central processing unit places the request on a queue and, when it is ready, begins to execute the device driver
17. The device driver determines which command has successfully completed, removes it from its queue and informs the operating system that its should execute the network interface device driver because something that interests it has happened

18. The network interface device driver determines which command has successfully completed, removes it from its queue and informs the rest of the network stack that the page arrived until finally the page fault handler is informed that its page is now located in main memory
19. The page fault handler updates the page table to reflect the new location of the page and informs the central processing unit that it can resume executing the application that caused the page fault

This sequence does not appear to be any more complicated than the previous one. Appearances are deceptive because steps 4 through 8 have been understated. Exposing these issues requires an understanding of networks.

A network is a collection of computers joined by a communication link and a common protocol that allows them to successfully transmit messages from one to another. Because the communication link is shared, a computer that wishes to send a message to another must usually wait it turn or ask to be given permission to speak. If this were not so, then most messages would collide with other messages traveling along the network and very little useful communication would take place. As more computers are added to a network, the amount of time that they must wait to send a message relative to the amount of time needed to transmit the message quickly increases. Some relief can be obtained by limiting the length the message each computer can send before it must yield the network to another machine by splitting long messages into a sequence of smaller *packets*. Even then, if too many computers are connected to the network the wait becomes intolerable.

To allow large (*wide-area*) networks to work, the network is divided into many smaller networks called *subnets*. Each subnet is limited to a handful of computers. This makes it easy for one computer to communicate with another computer on the same subnet without having to wait very long. Within each subnet there is a special computer known as a *gateway*. The gateway is special in that it is able to communicate with the world beyond the subnet. When a computer needs to send a message to a computer on another subnet, it sends it to the gateway. The gateway receives the message and decodes it enough to determine who the intended receiver is. The gateway consults a data structure called a *routing table* to determine which of the computers that it can communicate with can forward the message to its intended receiver. The process of receiving a message, consulting the routing table and forwarding the message is called a *hop*. Sometimes, the gateway will receive a message from beyond its subnet addressed to one of the subnet computers. The gateway will realize that it can communicate with that computer directly and forwards the message straight to the destination computer. All network interfaces on the subnet will "see" the message, but only the network interface on destination computer informs its network stack that a message has arrived.

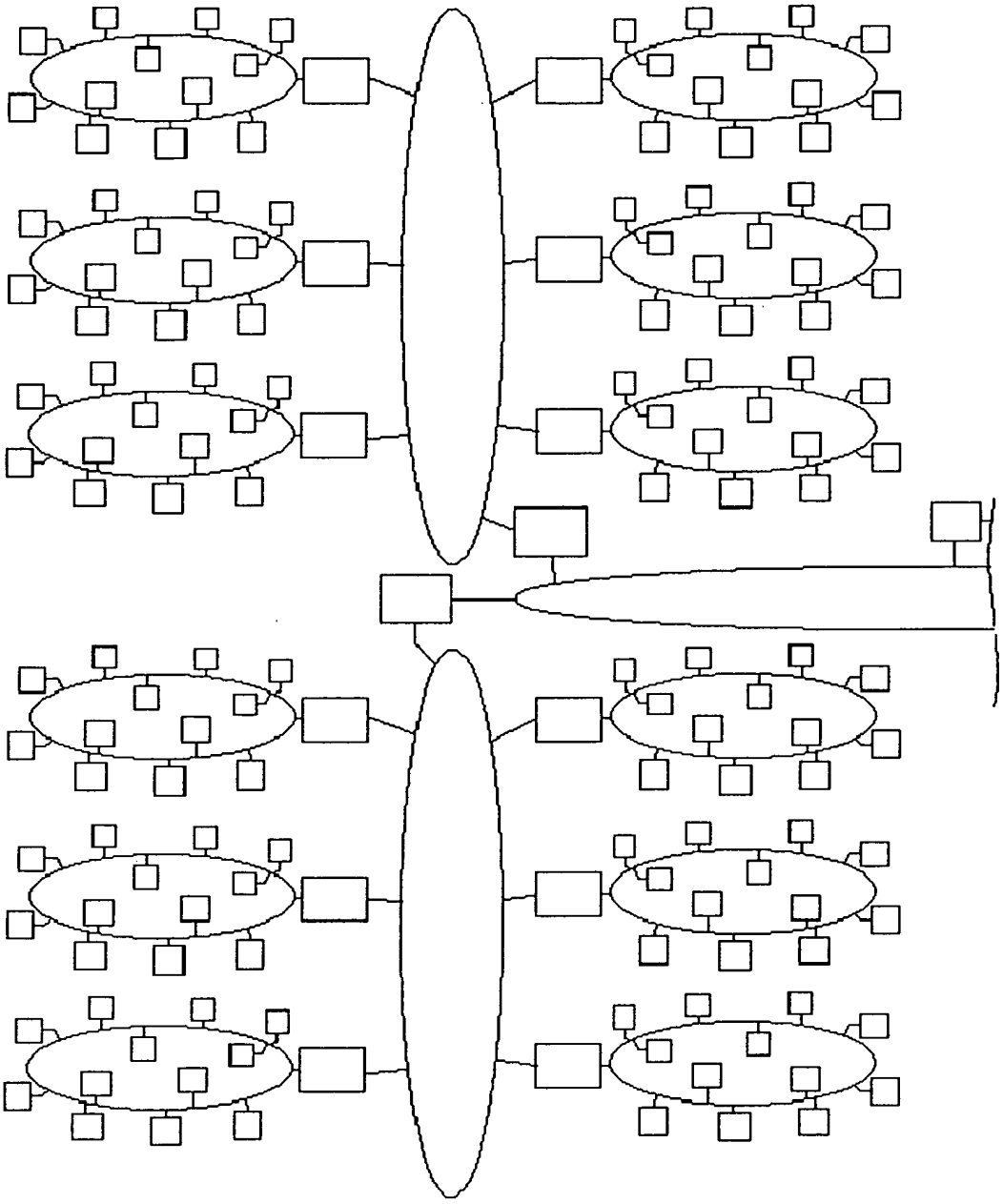


Figure 3: Portion of a Wide-Area Network

Figure 3 illustrates how a large (*wide-area*) network such as the internet can be constructed by subdividing the network into many subnets and linking them together using gateways and other special computers, called *routers*, that do nothing but exchange messages. Using this scheme, a lot of communication transpires in parallel on different subnets so that each individual machine does not have to wait long to send a message on its subnet. The downside of this scheme is that messages might need to cross many subnets to reach their destination. At each crossing, a hop takes place requiring the address of the message to be decoded, a decision to be made about which computer to forward the message along to, and a forwarding of the message on a different subnet. Along the way, different subnets might impose different limits regarding how long each

message can be which may force a gateway or router to split the message up into two or more packets and forward each of them separately.

With this understanding of networks, we return to steps 4 and 5 of the network paging process. If the computer containing the page is on the same subnet as the computer running the application, these steps take little time to complete. Depending on the type of network, its bandwidth and assuming that the distance between the computers is no more than a few hundred meters, the latency of this event is in the 0.1 to 1 msec range. These assumptions hold for small, carefully crafted commercial environments. Step 6 requires the receiving computer to process the request for the page through its network stack, locate the requested page and invoke its network stack to send the page data. The time required to locate the page will likely be similar to the time it takes to obtain a page from a local hard disk. Sending the reply involves another short delay to obtain clearance to use the network. If the network has been crafted properly, as would be the case in a commercial subnet, then the reply will not have to be split up into smaller packets. Step 7 depends on the bandwidth of the network. Assuming commercial, 100 Mb/sec bandwidth, this will take 0.3 msec.

Given a carefully chosen and configured subnet as one might expect to be able to craft in a commercial environment, each page reference over the network would be serviced in no more than 15 msec. The large application that incurs 1000 page faults will spend 15 seconds waiting for pages. This is nearly indistinguishable to a human from the local hard disk and quite acceptable. A commercial subnet environment is also easy to isolate and protect from potential security hazards with *firewall* and *proxy* gateway computers that allow only trusted messages to enter and leave the subnet. This level of security ensures that the application cannot be obtained without permission and that computers on the subnet cannot be improperly controlled by replacing real page reply messages with pages containing a dangerous *virus* or *Trojan horse.*

Consider the case where the computer executing the application and the computer serving the pages are several network hops from each other and connected across subnets whose bandwidth is less than 100 Mb/sec. Each hop incurs a delay of 1 to 10 msec while the message address is decoded, a routing table consulted and the message buffered and re-packaged to send on to the next hop. Some subnets, particularly the ones that reach a residence are physically large (many miles between the gateway and the other computers) and have bandwidths of between 0.5 Mb/sec to 2 Mb/sec. Under these conditions a page request could be expected to take anywhere from 60 msec to upwards of 600 msec to be serviced, or 5 to 50 times the local hard disk page service interval. In this scenario, a large application making 1000 page requests would spend 60 to 600 seconds waiting for pages. This is a very noticeable and unacceptable amount of time. Yet, it is this scenario much more than the previous one that is reflects the environment available to wide-area commercial network and residential users.

Technical Description of the Invention

The invention consists of the following components:

1. an *execution controller:* Run when an application resident on a remote system (called the *server*) is to be launched on a local system (called the *client*). Establishes association between application to be run on the client and its associated files on the server. Handles client side of initial security protocol between client and server.
2. an *application remote file interface:* Handles client interface to accessing files associated with an application that is resident on a remote server.
3. an *application cache manager:* On the client, locally stores previously requested portions of files and file system information associated with an application resident on a remote server. Requests referenced application code and data not currently in the client's cache. Employs existing profiling information to prefetch portions of the application from the server. Collects new profiling information about application to improve client prefetching in the future; this profiling information may also be uploaded to the server for use in improving prefetching performance and to assist in better pre-compression of file data.
4. an *application file server:* Responds to requests by client application cache manager for portions of application's files and directory structure on the server. Transmits compressed information (which may be pre-compressed with nearby data) for better bandwidth utilization. May send extra file data beyond that requested, if that data is expected to be referenced soon.
5. a *file system reference profile processor:* Processes the uploaded sequence of application file references and frequency information. This information, which was collected by the application cache manager, is used in its processed form by the "application stream set builder" in generating pre-compressed file datasets.
6. an *application stream set builder*, used to construct the *application stream sets* that the application file server consults to reply to application file requests.

The execution controller is a small piece of code that resides on the client. The execution controller is given an *argument* indicating which application is to be executed. From the point of view of the client and its operating system, the application is resident locally on the client; the execution controller negotiates with an appropriate server to allow the client to obtain (as needed) segments of the associated application files located on the server.

The execution controller handles the client side of the security protocol between the client and a server; one approach to implementing this security protocol is as follows. The execution controller contains a *security certificate* which uniquely identifies/distinguishes it from every other instance of an execution controller. This certificate has a *private key* that can be used to encrypt any message so that it can be decrypted only with the corresponding *public key* known to the server. Additionally, the execution controller knows the public key of the server but only the server has the private key which can decrypt messages encrypted with it. When starting, the execution controller forms a message indicating which application it is instructed to execute and

attaches to this message a randomly generated key which will be used to encode all subsequent messages between the client and server. The client encrypts this message with its private key and then encrypts the encrypted message with the server's public key. This doubly encrypted message is sent across the network to the server. The server uses its private key to decrypt the message. This has the effect of giving the client a high degree of confidence that only the true server intended to receive the encrypted message views its actual contents. The server then decrypts the message again using the client's public key. This has the effect of giving the server a high degree of confidence that the message was generated by the client. As a result, the server knows which application the client wants to execute and which random key to use in subsequent exchanges with the client.

Upon receiving notice from a client that it wants to execute an application, a server (or set of servers) performs the following actions:

- consults a database which indicates which applications the client is allowed to execute and, if the client is not allowed to execute the requested application, informs the client that it will not be served,
- determines if the application has been updated and, if it has, indicates this to the client, along with information concerning accessing the updated version,
- determines appropriate server location(s) of the desired application,
- checks load on these candidate servers,
- refers the client to the candidate server with the most appropriate load,
- informs the client that it is ready to serve the application.

Upon receiving a reply, the client will either continue with the application execution process, notify the user that it cannot proceed, or interact with the user to determine what action to pursue next, depending on the nature of the reply returned by the server.

If a server accepts the task of serving the application to the client, the execution controller passes the application access request on to the application remote file interface code. This code allows the client to reference file and directory information associated with the remote application as if it resided on a local physical disk device. It uses the network stack to request portions of the application's files and directory structure from the server and borrows storage space from a physical hard disk device on the client to archive this data for future use. The archive storage on the client is managed by the application cache manager, which is another small piece of code running on the client. The invention requires the execution controller, the application remote file interface, and the application cache manager to have been previously installed on the client via traditional software delivery methods.

The client's operating system begins executing the requested application located remotely on a server. The operating system memory-maps the application and begins executing it, with the application remote file interface code obtaining control whenever the client system's page fault handler determines that the application's page is located on the remote disk drive. The page fault handler asks the application remote file interface code to place the appropriate page data in main memory. The application remote file interface code sends a request to the cache manager for the desired data. If the application cache

manager has the data, it is placed in main memory and the application remote file interface code returns control to the page fault handler. If the application cache manager does not have the requested page data, it formulates a message to the server indicating which portions of the remote disk it needs, encrypts this message with the random key that the execution controller produced, and invokes the network stack to send the read message to the server. The requested portion of file data is identified by file name (or some numeric ID) and the pages of the file desired.

The application file server, upon receiving the message, decrypts it with the random key. This gives both client and server confidence that the request was sent by the real client and received by the real server. The server uses the file name and portion of the requested application to lookup or create a reply message. The simplicity of this action is critical to the invention because it is essential that the server respond quickly to any page request. The server makes every effort to index and pre-compute reply messages and to keep them in main memory where they can be rapidly accessed by the server's central processing unit. The response message may contain not only the requested page data, but also several other pages that will very likely be needed by the client in the near future. Alternatively, the client itself may request pages in advance of the application demanding them, by use of its local profile data. The stored response message is also pre-compressed to reduce its length; it is expected to be approximately halved. The response message is encrypted with the random key (this step is not done in advance, since each client sends a different random key, which is used instead of private and public key pairs because they require less time-intensive algorithms) and sent back to the client.

Upon receiving the reply, the client decrypts the message using the random key. This gives the client a high degree of confidence that it is receiving the reply sent by the real server. The client un-compresses the response and parses out the pages returned in the reply. Each page is returned to the application cache manager for future reference. The requested page is placed in main memory and the application remote file interface code returns control to the page fault handler, which allows the client's central processing unit to proceed executing the application.

When the next page fault occurs, there is a high probability that the application cache manager already holds the requested page. The page was either sent by the server on a previous run of the application or was packaged in a previous response to a page request during the current run. This likelihood is because applications have a significant amount of predictability in the order to which they reference sectors on a disk. These patterns can be determined over time by keeping a trace of page requests. In the invention, the application cache manager performs this task. As requests for pages are sent to the cache manager, it notes which pages were previously referenced in a table indexed by the page number. For example, when a request for page 513 of some file is followed by a request for page 1023 of some file, the cache manager records this information in a *page trace table*. The information in this table may be compiled into a message and sent to the server periodically and upon exit of the application. This process, known as sampling, places very little computational demand on the client and the server. The server stores these

tables and uses them after some time to build a new application stream set for the application.

Aggregation and analysis of the uploaded profile data is done by the file system reference profile processor, which is an application executed on a computer system that may be different than the client or server. The following process may be employed by this code to produce trace data used to build application stream sets:

- initialize a two-dimensional table, a, from $a[0, 0]$ to $a[s_{max}, s_{max}]$, where s_{max} is the largest page number, to zeroes,
- for every element $t[s]$ in a page trace table, t, where $t[s]$ is a valid page add one to $a[t[s], s]$ and
- for every column c in a, calculate d to be the sum $a[c, 0] + a[c, 1] + ... + a[c, s_{max}]$ and, if d is not zero, divide every element of the column by d.

The effect of this process is to generate a table a, where $a[s, f]$ indicates the probability with which page f has been measured to follow page s. This will be 0 if f was never found to follow s and 1.0 if f was always found to follow s. Probability theory dictates that, given a sufficiently large set of page trace tables, the probabilities in a will be very close approximations of the actual probabilities.

The process of building a new set of request replies for an application is called building an application stream set. This process takes place on a computer that may be different than the client or the server and takes place at least once before the application is executed using the invention. An application stream set contains:

- a unique name of the application for reference purposes,
- an index table used to quickly determine which reply to return for a given request,
- the set of all possible request replies, each one being a catenation of the actual page requested followed by zero or more additional pages that are deemed by the application stream set building algorithm to be highly likely to be referenced immediately following the first reference, the collection of which is then compressed using a suitable compression algorithm.

The application stream set is built in the following manner:

- instantiate a virtual hard disk drive large enough to contain all of the application's executable and non-executable data and all of the indices (often known as *directories* and *files*) needed by the operating system to properly identify and reference the data,
- install the application on the virtual hard disk using any one of the traditional application delivery models,
- initialize the application stream set to be empty,
- add the unique name of the application to the stream set,
- add an index table to the application stream set containing an entry for each sector in the virtual hard disk drive,
- for each page, s, in the virtual hard disk drive:
    - initialize a buffer to be empty,
    - place the page data of s in the buffer,

- if aggregated page trace data, a, is not available, skip the following step and go to the compression step,
- perform the following sub-steps:
- initialize set m to contain the pair $\{s, 1.0\}$,
- for some pair $\{s_0, p0\}$ in m, if $p_0 \times a[s_0, s_1]$ is greater or equal to threshold t, and s_1 is not already in m, add $\{s_1, p_0 \times a[s_0, s_1]\}$ to m,
- add a fixed-sized marker indicating the number of s_1 to the buffer,
- add the page data of s_1 to the buffer,
- repeat the previous three sub-steps until no more items can be added to m,
- compress the data in the buffer and add it to the application stream set and
- update the index table entry corresponding to the page to reference the starting location of the just-added compressed data buffer.

The process of building an application stream set must be started without any aggregated trace data since the trace data cannot be collected until there is an application stream set with which to execute the application. The process that is followed is to build an application stream set using the process given without aggregated trace data. This will result in an application stream set that contains only one page per page reply. This application stream set is then used in a controlled commercial subnet environment so that the application will execute with reasonable performance. This environment is used to execute the application under normal conditions for several hours. This will yield enough trace tables to produce the first cut of aggregated trace data that will yield an application stream set that allows the application to execute across a less controlled network environment. This new application stream set can then be used for enough time to collect a much greater set of trace tables which in turn will allow an even better application stream set to be built. This process can be iterated several times.

The most appropriate value of threshold t varies for each different application. Too high a threshold value (near 1.0) will result in responses that contain few pages in each response message and will not improve the performance of the invention over a simple service method. Too low a threshold value (near 0.0) will result in replies that contain too many pages and will require too long to be sent. Such replies will cause the paging response performance of the application to be erratic and noticeable to the client's user. The ideal reply size for the network connection under consideration can be determined via analysis or experimentation. Then the application stream set builder can automatically determine the most appropriate threshold value t using a simple binary search technique. The builder starts with a threshold of 0.5 and builds an application stream set. If the average number of sectors in each reply is greater than that desired, then it subtracts 0.25 from the threshold value and iterates through the build process. If the average number of sectors in each reply is lower than desired, then it adds 0.25 to the threshold and iterates through the build process. The iterative process continues with the amount added or subtracted reduced in half on each iteration. The process ends when either the desired reply size is reached or when a large number, say 100, build iterations have transpired.

Benefits of the Invention

The first benefit of the invention is a dramatic reduction in the perceived paging delay when operating across a network. By choosing appropriately sized request replies that have a high probability of proximate reference, each response returns several useful pages for the latency of one. Compressing the replies to reduce their average length in approximately half effectively doubles the bandwidth of the network. Together, these strategies yield a substantial reduction in the perceived latency. Thus, an unacceptable delay of (say) 60 seconds becomes an acceptable delay of (say) 12 seconds. Additionally, by automatically caching returned data, the invention nearly eliminates the need for network requests on all but the first execution of the application. After an initial, slightly slower than average execution, the application will generally execute with the same paging behavior as if it were traditionally delivered on the client.

Performance measurements collected using an implementation of the invention strongly demonstrate its value. The wall-clock time required to execute the Microsoft Office Word application (bring it up and shut it down) across a 1 Mbps link with the naïve network-unaware approach of no prefetching and no compression, and with no client cache is 47.6 seconds. The wall-clock time required to execute Word across a 1 Mbps link with prefetching based on profile data collected from a previous run enabled, compression enabled, and a completely empty client cache is 19.4 seconds. This greater than 50% reduction in execution time illustrates the gain due to fetching accurately predicted pages in advance along with compression of the set of pages together. And finally, the wall-clock time required to execute Word across a 1 Mbps link with prefetching based on profile data collected from a previous run enabled, compression enabled, and a cache warmed by a previous run is 4.0 seconds. This additional improvement shows that persistent caching of application file pages brings performance very close to native on subsequent runs, with minimal network load. [The latter two runs include a compression strategy reducing the bits transferred by about 40%.]

Through the use of security certificates and randomly generated keys, the invention provides a high level of security and confidence across public networks. The randomly generated key also reduces the amount of computation required to encrypt and decrypt application data while maintaining sufficient security to operate across an open network. To provide additional security to the application provider, the application stream set can be built with randomly positioned *land-mine sectors* that are associated with the application but would never be referenced during normal execution. If a *cracker* were to wrest control of the virtual device from the execution controller on a client and attempt to make a copy of the installed application, the client would request a land-mine sector which would alert the server that an act of software piracy was being attempted. The server then may choose to deny all requests from the client until the matter is properly investigated.

By providing instant execution of applications across a public network, the invention engenders new revenue models for software developers and new usage models for software consumers. Software developers can allow customers to demo or *test drive* their application in hopes of enticing more customers to buy the application. Software developers can charge per use of an application, based on either the number of times the

application is executed or by the amount of time actually spent executing the application. Software consumers also benefit because they can use their applications from any suitable computer system attached to the network. Traditional software delivery models make this very inconvenient because the consumer must carry with them the physical media containing the application and must often go through the process of un-installing the application to abide by the application's *software license agreement.*

Prior Art

US Patent 5,790,753: System for downloading computer software programs
US Patent 5,781,226: Network virtual memory for a cable television settop terminal

Exhibit C10

eStream AppInstallBlock Low Level Design

Sanjay Pujare and David Lin

Version 0.2

[redacted]

Functionality

The AppInstallBlock is a block of code and data associated with a particular application. This AppInstallBlock contains the information needed to by the eStream client to 'initialize' the client machine before the eStream application is used for the first time. It also contains optional profiling data for increasing the runtime performance of that eStream application.

The AppInstallBlock is created offline by the eStream Builder program. First of all, the Builder monitors the installation process of a local version of the application installation program and records changes to the system. This includes any environment variables added or removed from the system, and any files added or modified in the system directories. Files added to the application specific directory is not recorded in the AppInstallBlock to reduce the amount of time needed to send the AppInstallBlock to the eStream client. Secondly, the Builder profiles the application to obtain the list of critical pages needed to run the application initially and an initial page reference sequence of the pages accessed during a sample run of the application. The AppInstallBlock contains an optional application-specific initialization code. This code is needed when the default initialization procedure is insufficient to setup the local machine environment for that particular application.

The AppInstallBlock and the runtime data are packaged into the eStream Set by the Builder and then uploaded to the application server. After the eStream client subscribed to an application and before the application is run for the first time, the AppInstallBlock is send by the server to the client. The eStream client invokes the default initialization procedure and the optional application-specific initialization code. Together, the default and the application-specific initialization procedure process the data in the AppInstallBlock to make the machine ready for eStreaming that particular application.

Data type definitions

The AppInstallBlock is divided into the following sections: header section, variable section, file section, profile section, prefetch section, comment section, and code section. The header section contains general information about the AppInstallBlock. The information includes the total byte size and an index table containing size and offset into other sections. In Windows version, the variable section consists of two registry tree structures to specify the registry entries added or removed from the OS environment. The file section is a tree structure consisting of the files copied to C drive during the application installation. The profile section contains the initial set of block reference sequences during

Builder profiling of the application. The prefetch section consists of a subset of profiled blocks used by the Builder as a hint to the eStream client to prefetch initially. The comment section is used to inform the eStream client user of any relevant information about the application installation. Finally, the code section contains an optional program tailored for any application-specific installation not covered by the default eStream application installation procedure. In Windows version, the code section contains a Windows DLL.

Here is a detailed description of each fields of the AppInstallBlock.

Note: Little endian format is used for all the fields spanning more than 1 byte. Also, BlockNumber specifies blocks of 4K byte size.

1. Header Section:

The header section contains the basic information about that AppInstallBlock. This includes the versioning information, application identification,

Core Header Structure:

- **AibVersion [4 bytes]**: Magic number or appInstallBlock version number (which identifies the version of the appInstallBlock structure rather than the contents).
- **AppId [16 bytes]**: this is an application identifier unique for each application. On Windows, this identifier is the GUID generated from the 'guidgen' program. AppId for Word on Win98 will be different from Word on WinNT if it turns out that Word binaries are different between NT and 98.
- **VersionNo [4 bytes]**: Version number. This allows us to inform the client that the appInstallBlock has changed for a particular appId. This is useful for changes to the AppInstallBlock due to minor patch upgrades in the application.
- **ClientOSBitMap [4 bytes]**: Client OS supported bitmap or ID: for Win2K, Win98, WinNT and other future OSs we might support (it should be possible to say that this appInstallBlock is for more than one OS).
- **ClientOSServicePack [4 bytes]**: We might want to store the service pack level of the OS for which this appInstallBlock has been created. Note that when this field is set we cannot use multiple OS bits in the above field ClientOSBitMap.
- **Flags [4 bytes]**: Flags pertaining to AppInstallBlock
    - **Bit 0:** Reboot – If set, the eStream client needs to reboot the machine after installing the AppInstallBlock on the client machine.
    - **Bit 1:** Unicode – If set, the string characters are 2 bytes wide instead of 1 byte.
- **HeaderSize [2 bytes]**: Total size in bytes of the header section.
- **Reserved [32 bytes]**: Reserved spaces for future.

- **NumberOfSections [1 byte]**: Number of sections in the index table. This determines the number of entries in the index table structure described below:

Index Table Structure: (variable number of entries)

- **SectionType [1 bytes]**: The type of data describe in section. 0=file section, 1=variable section, 2=prefetch section, 3=profile section, 4=comment section, 5=code section.
- **SectionOffset [4 bytes]**: The offset from the beginning of the file indicates the beginning of section.
- **SectionSize [4 bytes]**: The size in bytes of section.

Variable Structure:

- **ApplicationNameLength [4 bytes]**: Byte size of the application name
- **ApplicationName [X bytes]**: Non-null terminating name of the application

2. File Section:

The file section contains a subset of the list of files needed by the application to run properly. This section does not enumerate files located in the standard application program directory. It consists of information about files copied into 'unusual' directory during the installation of an application. If the file content is small, the file is copied to the client machine. Otherwise, the file is relocated to the standard program directory suitable for streaming. The file section data is list of trees stored in a contiguous sequence of address space according to the pre-order traversal of the trees. A node in the tree can correspond to one or more levels of directory. A parent-child node pair is combined into a single node if the parent node has only a single child. Parsing the tree from the root of the tree to a leaf node results in a fully legal Windows pathname including the drive letter. Each entry of the node in the tree consists of the following structure:

Directory Structure: (variable number of entries)

- **Flags [4 byte]**: Bit 0 is set if this entry is a directory
- **NumberOfChildren [2 bytes]**: Number of nodes in this directory
- **DirectoryNameLength [4 bytes]**: Length of the directory name
- **DirectoryName [X bytes]**: Non-null terminating directory name

Leaf Structure: (variable number of entries)

- **Flags [4 byte]**: Bit 1 is set to 1 if this entry is a spoof or copied file name
- **FileVersion [4? bytes]**: Version of the file GetFileVersionInfo() if the file is win32 file image. Need variable file version size returned by GetFileVersionInfoSize(). Otherwise use GetFileTime() to retrieve the file creation time.
- **FileNameLength [4 bytes]**: Byte size of the file name

- **DataLength [4 bytes]**: Byte size of the data. If spoof file, then data is the string of the spoof directory. If copied file, then data is the content of the file
- **FileName [X bytes]**: Non-null terminating file name
- **Data [X bytes]**: Either the spoof file name or the content of the copied file

3. Add Variable and Remove Variable Sections:

The add and remove variable sections contain the system variable changes needed to run the application. In Windows system, each section consists of several number of registry subtrees. Each tree is stored in a contiguous sequence of address space according to the pre-order traversal of the tree. A node in the tree can correspond to one or more levels of directory in the registry. A parent-child node pair is combined into a single node if the parent node has only a single child. Parsing the tree from the root of the tree to a leaf node results in a fully legal key name. The order of the trees is shown here.

a. Registry Subsection:

1. "KHCR": HKEY_CLASSES_ROOT
2. "HKCU": HKEY_CURRENT_USER
3. "HKLM": HKEY_LOCAL_MACHINE
4. "HKU": HKEY_USERS
5. "HKCC": HKEY_CURRENT_CONFIG

Tree Structure: (5 entries)

- **ExistFlag [1 byte]**: Set to 1 if this tree exist, 0 otherwise.
- **Key or Value Structure entries [X bytes]**: Serialization of the tree into variable number key or value structures described below.

Key Structure: (variable number of entries)

- **KeyFlag [1 byte]**: Set to 1 if this entry is a key or 0 if it's a value structure
- **NumberOfSubchild [4 bytes]**: Number of subkeys and values in this key directory
- **KeyNameLength [4 bytes]**: Byte size of the key name
- **KeyName [X bytes]**: Non-null terminating key name

Value Structure: (variable number of entries)

- **KeyFlag [1 byte]**: Set to 1 if this entry is a key or 0 if it's a value structure
- **ValueType [4 byte]**: Type of values from the Win32 API function RegQueryValueEx(): REG_SZ, REG_BINARY, REG_DWORD, REG_LINK, REG_NONE, etc...
- **ValueNameLength [4 bytes]**: Byte size of the value name
- **ValueDataLength [4 bytes]**: Byte size of the value data

- **ValueName [X bytes]**: Non-null terminating value name
- **ValueData [X bytes]**: Value of the Data

In addition to registry changes, an installation in Windows system may involve changes to the ini files. The following structure is used to communicate the ini file changes needed to be done on the eStream client machine. The ini entries are appended to the end of the variable section after the 5 registry trees are enumerated.

b. INI Subsection:

- **NumFiles [4 bytes]**: Number of INI files modified.

File Structure: (variable number of entries)

- **FileNameLength [4 bytes]**: Byte length of the file name
- **FileName [X bytes]**: Name of the INI file
- **NumSection [4 bytes]**: Number of sections with the changes

Section Structure: (variable number of entries)

- **SectionNameLength [4 bytes]**: Byte length of the section name
- **SectionName [X bytes]**: Section name of an INI file
- **NumValues [4 bytes]**: Number of values in this section

Value Structure: (variable number of entries)

- **ValueLength [4 bytes]**: Byte length of the value data
- **ValueData [X bytes]**: Content of the value data

4. Prefetch Section:

The prefetch section contains a list of file blocks. The Builder profiler determines the set of file blocks critical for the initial run of the application. This data includes the code to start and terminate the application. It includes the file blocks containing for frequently used commands. For example, opening and saving of documents are frequently used commands and should be prefetched if possible. Another type of blocks to include in the prefetch section is the blocks associated with frequently accessed directories and file metadata in this directory. The format of the data is described below:

- **FileNumber [4 bytes]**: File Number of the file containing the block to prefetch
- **BlockNumber [4 bytes]**: Block Number of the file block to prefetch

5. Profile Section: (not used in eStream 1.0)

The profile section consists of a reference sequence of file blocks accessed by the application at runtime. Conceptually, the profile data is a two dimensional matrix. Each entry [*row, column*] of the matrix is the frequency a block *row* is followed by a block *column*. In any realistic applications of fair size, this matrix is very large and sparse. Proper data structure must be selected to store this sparse matrix efficiently in required storage space and minimize the overhead in accessing this data structure access.

The section is constructed from two basic structures: row and column structures. Each row structure is followed by N column structures specified in the NumberColumns field. Note that this is an optional section. But with appropriate profile data, the eStream client prefetcher performance can be increased.

Row Structure: (variable number of entries)

- **FileNumber [4 bytes]**: File Number of the row block
- **BlockNumber [4 bytes]**: Block Number of the row block
- **NumberColumns [4 bytes]**: number of blocks that follows this block. This field determines the number of column structures following this field.

Column Structure: (variable number of entries)

- **FileNumber [4 bytes]**: File Number of the column block
- **BlockNumber [4 bytes]**: Block Number of the column block
- **Frequency [4 bytes]**: frequency the row block is followed by column block

6. Comment Section:

The comment section is used by the Builder to describe this AppInstallBlock in more detail.

- **Comment [X bytes]**: Null terminating comment string

7. Code Section:

The code section consists of the application-specific initialization code needed to run on the eStream client to setup the client machine for this particular application. This section may be empty if the default initialization procedure in the eStream client is able to setup the client machine without requiring any application-specific instructions. On the Windows system, the code is a DLL file containing two exported function calls: *Install*(), *Uninstall*(). The eStream client loads the DLL and invokes the appropriate function calls.

- **Code [X bytes]**: Binary file containing the application-specific initialization code. On Windows, this is just a DLL file.

8. LicenseAgreement Section:

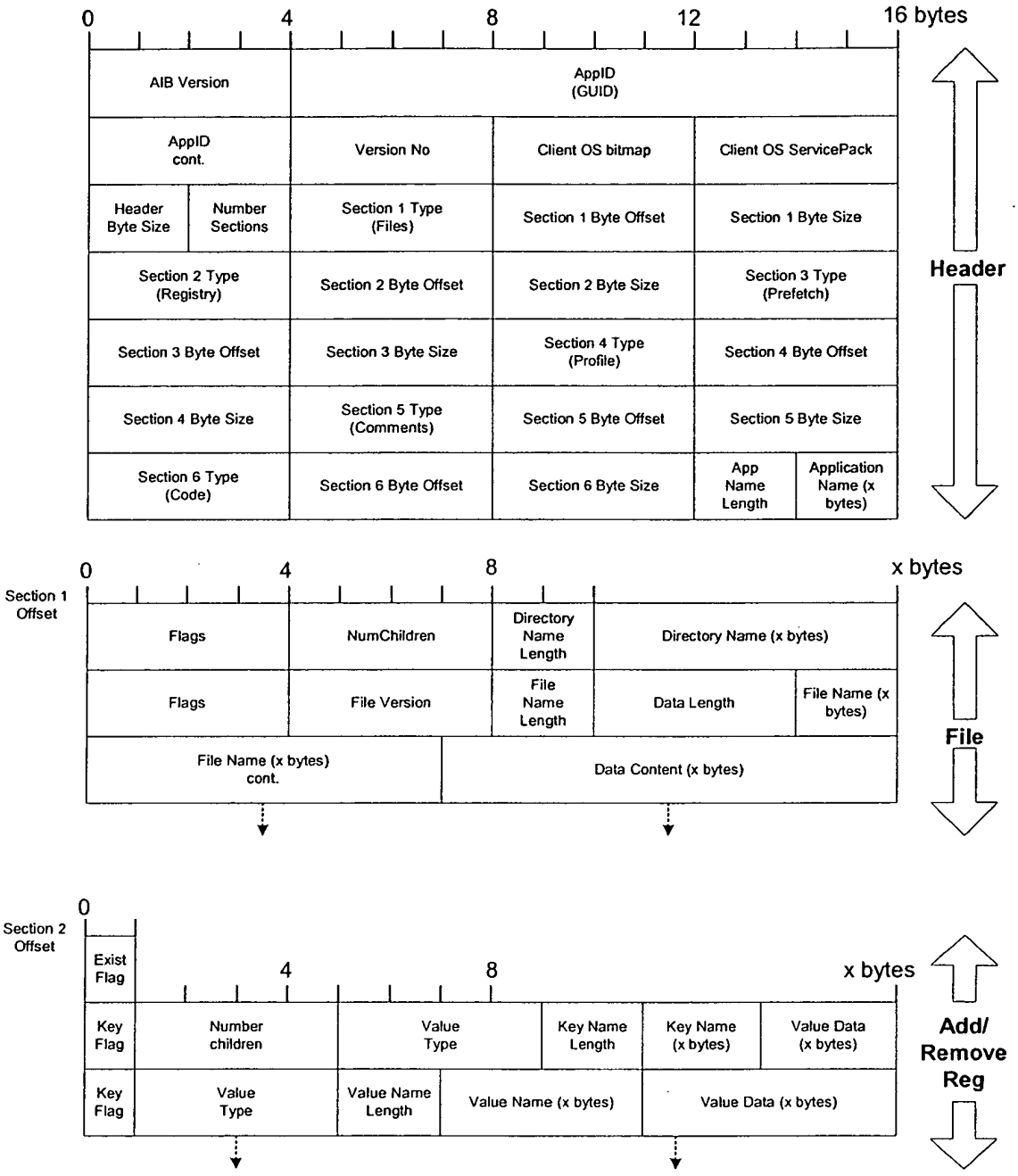
The Builder creates the license agreement section. The eStream client displays the license agreement text to the end-user before the application is started for the first time. The end-user must agree to all licensing agreement set by the software vendor in order to use the application.

- **LicenseAgreement [X bytes]**: Null terminating license agreement string

Open Issues

- What is the size of the AppInstallBlock for a typical application like Office?
- How large should the prefetch sections be for optimal run of an application? At minimum, it should contain at least start/termination code.
- How should the AppInstallBlock handle application license agreement text string? Add a new section or use comment section. Does the dialog need to have exactly the same interface as the license agreement dialog on the local installation?
- Currently, file section stores complete pathname including the drive letter. The installation may place files according to some variables like %SystemRoot% or %UserProfile%. How does the Builder detect this so it can propagate this information to the client?

Format of AppInstallBlock (part 1 of 2)



Format of AppInstallBlock (part 2 of 2)

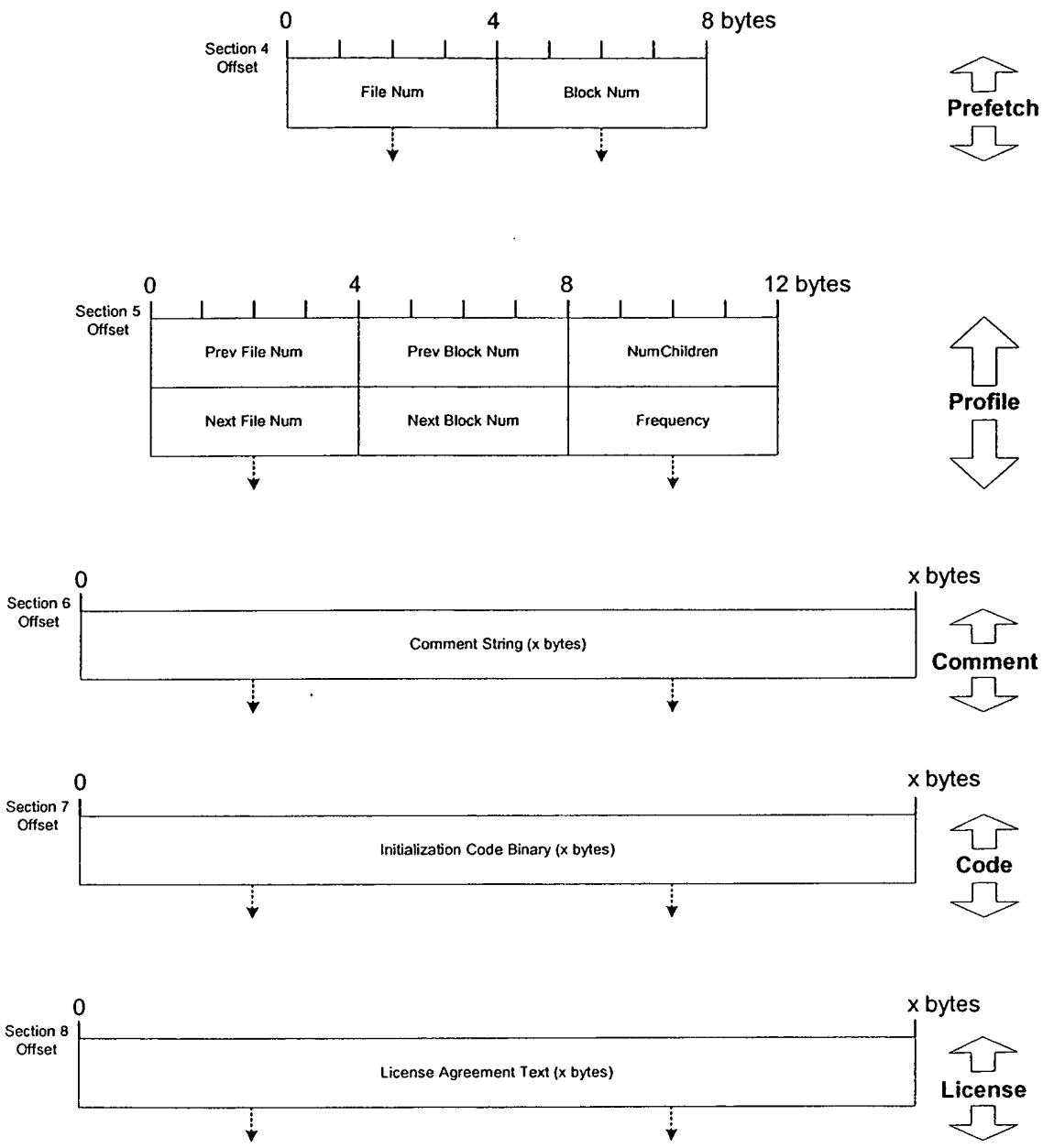
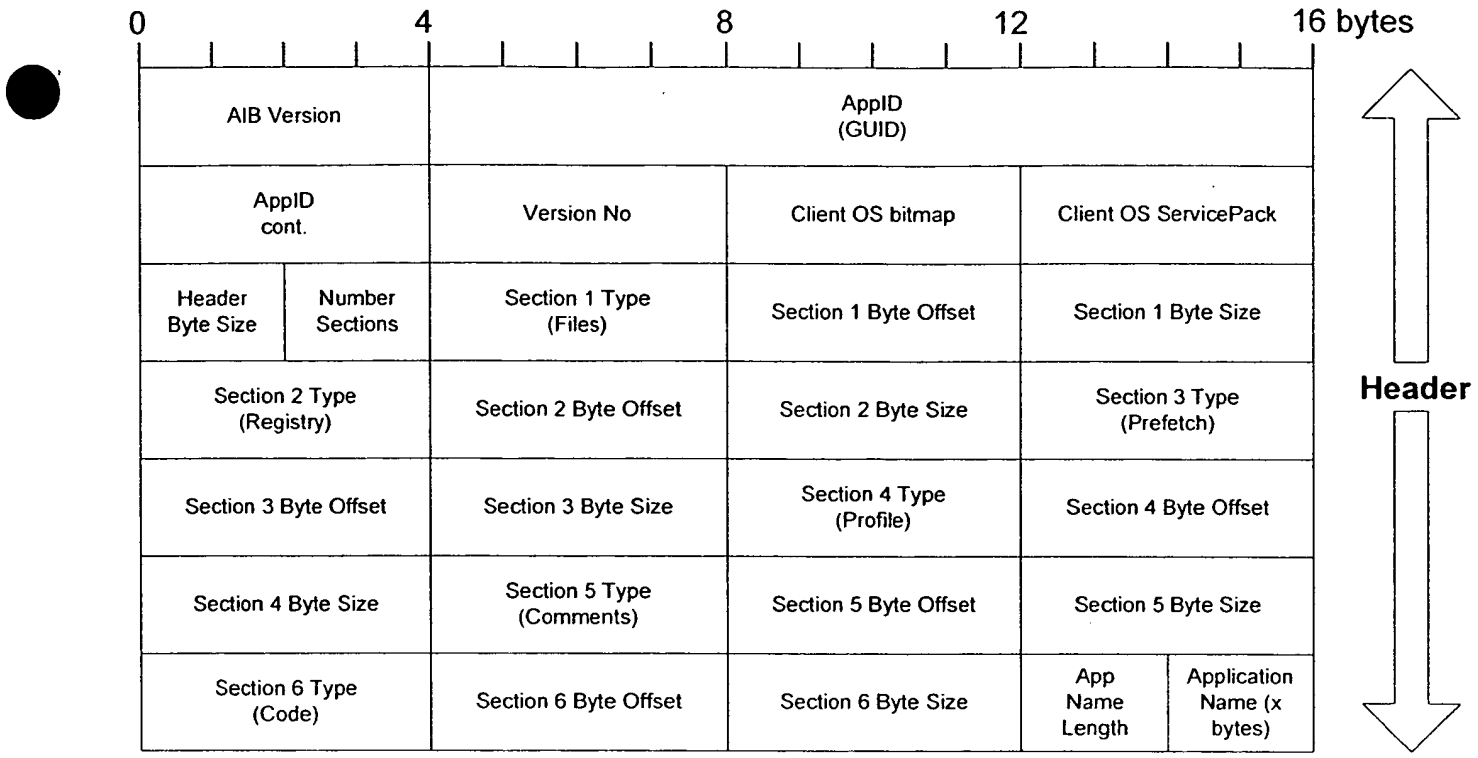
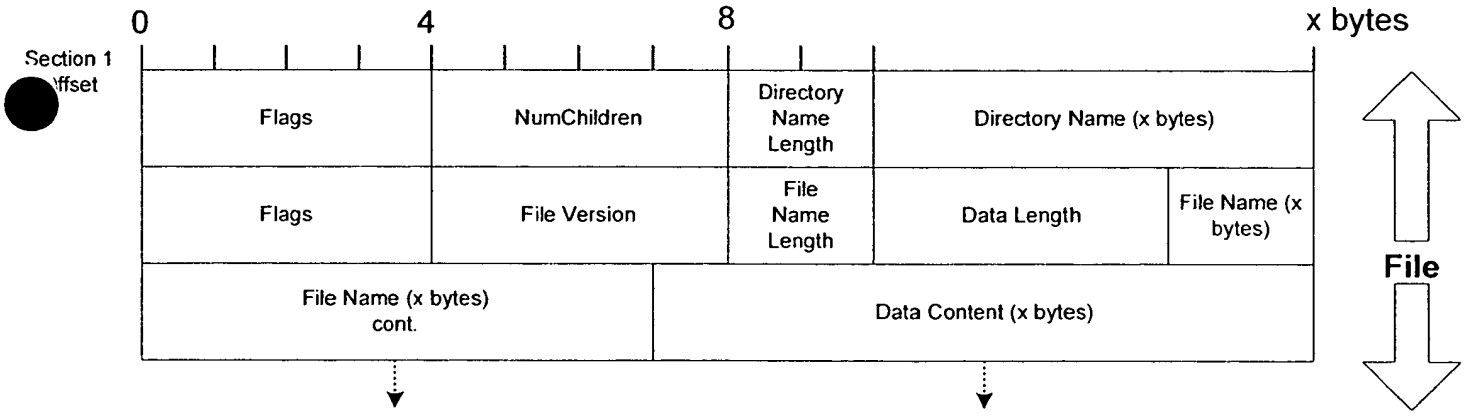
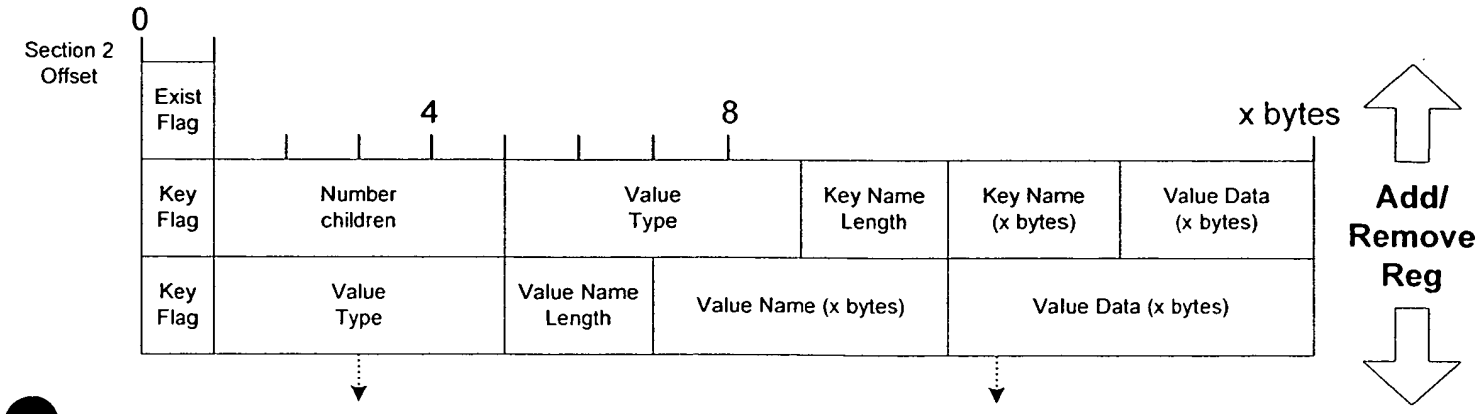


Exhibit C11

Format of AppInstallBlock (part 1 of 2)







Format of AppInstallBlock (part 2 of 2)

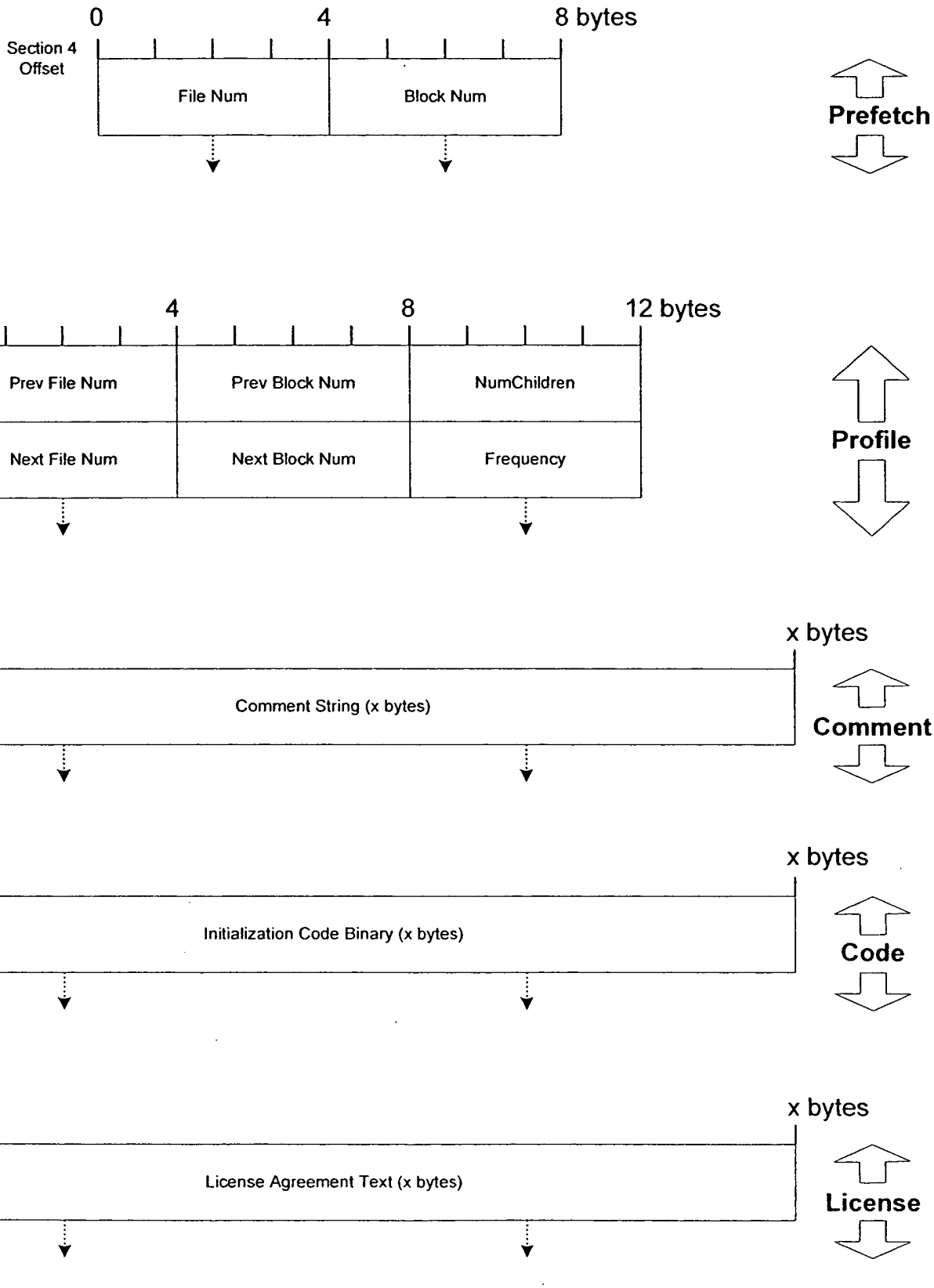


Exhibit C12

The eStream Builder

The eStream Builder is a software program. It is used to convert locally installable applications into a data set suitable for streaming over the network. The streaming-enabled data set is called the eStream Set. This document describes the procedure used to convert locally installable applications into the eStream Set.

The application conversion procedure into the eStream Set consists of the several steps. In the first phase, the Builder program monitors the installation process of a local installation of the desired application for conversion. The Builder monitors any changes to the system and records those changes in an intermediate data structure. After the application is installed locally, the Builder enters the second phase of the conversion. In the second step, the Builder program invokes the installed application executable and obtains sequences of frequently accessed file blocks of this application. Both the Builder program and the eStream client software use the sequence data to optimize the performance of the streaming process. Once the sequencing information is obtained, the Builder enters the final phase of the conversion. In this step, the Builder gathers all data obtained from the first two phase and processes the data into the eStream Set.

In the next sections, detailed descriptions of the three phases of the Builder conversion process are described. The three phases consists of installation monitoring, application profiling, and finally eStream packaging. In most cases, the conversion process is general and applicable to all type of system. In places where the conversion is OS dependent, the discussion is focused on Microsoft Windows environment. Issues on conversion procedure for other OS environment are described in later sections.

Installation Monitoring

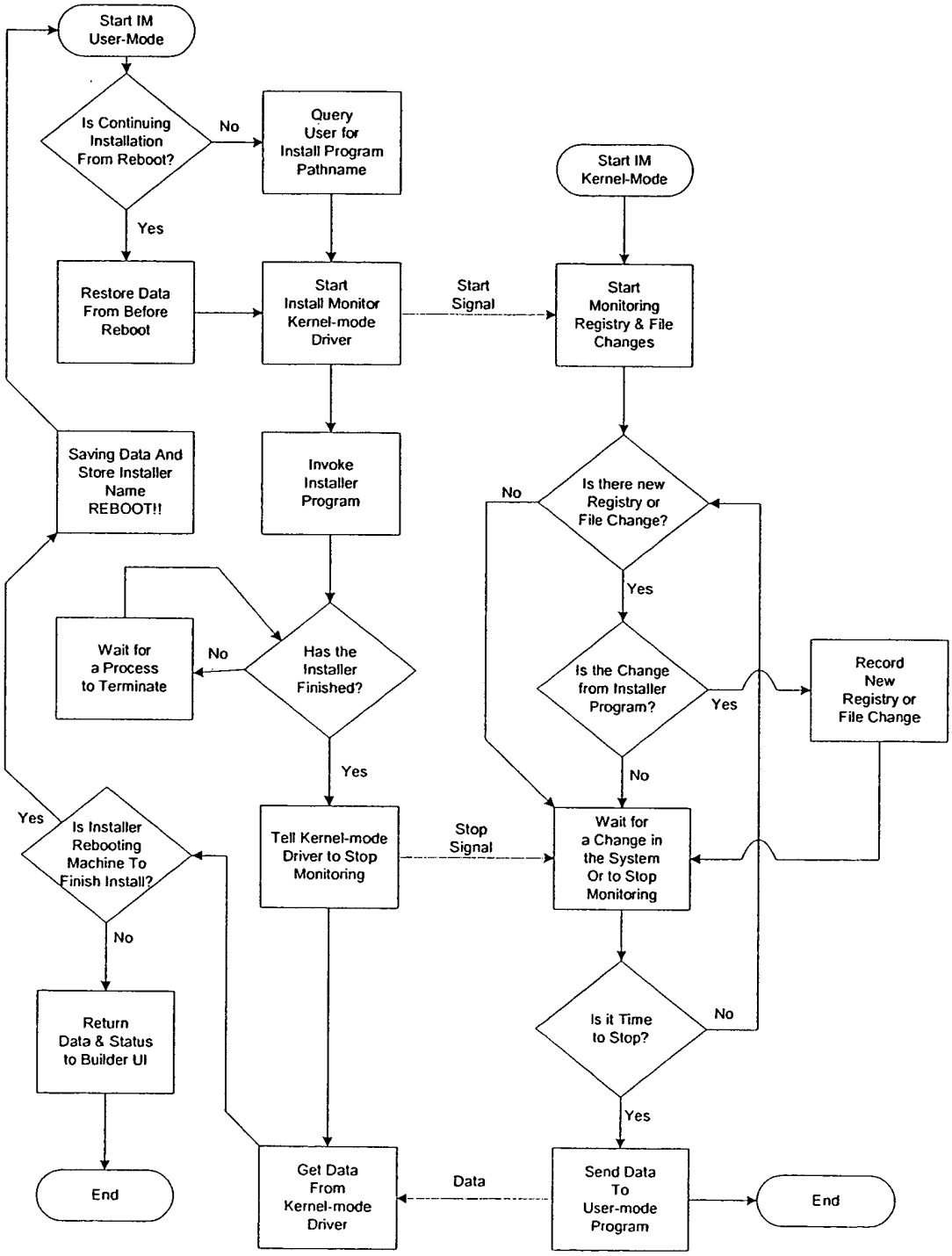
In the first phase of the conversion process, the Builder Installation Monitor (IM) component invokes the application installation program that installs the application locally. The IM observes all changes to the local computer during the installation. The changes may involve one or more of the following: changes to system or environment variables; and modifications, addition, or deletion of one or more files. The IM records all changes to the variables and files in a data structure to be sent to the Builder's eStream Packaging component. In the following paragraphs, detailed description of the Installation Monitor is described for Microsoft Windows environment.

In Microsoft Windows system, the Installation Monitor (IM) component consists of a kernel-mode driver subcomponent and a user-mode subcomponent. The kernel-mode driver is hooked into the Windows registry and file system function interface calls. The hook into the registry function calls allows the IM to monitor system variable changes. The hook into the file system function calls enables the IM to observe file changes.

The IM kernel-mode (IM-KM) driver subcomponent is controlled by the user-mode subcomponent (IM-UM). The IM-UM sends messages to the IM-KM to start and stop the monitoring process via standard I/O control messages called IOCTL. The IM-KM memorizes any addition or deletion of registry variables. It also records changes to

application-specific, shared among a group of applications, or system-wide files. Every files and directories are assigned a unique file number for simplifying identification of a specific file. Once the installation of an application completed, the IM-UM retrieves these changes from the IM-KM and forward the data structure to the eStream Packager.

Builder Install Monitor Control Flow Diagram



Application Profiling

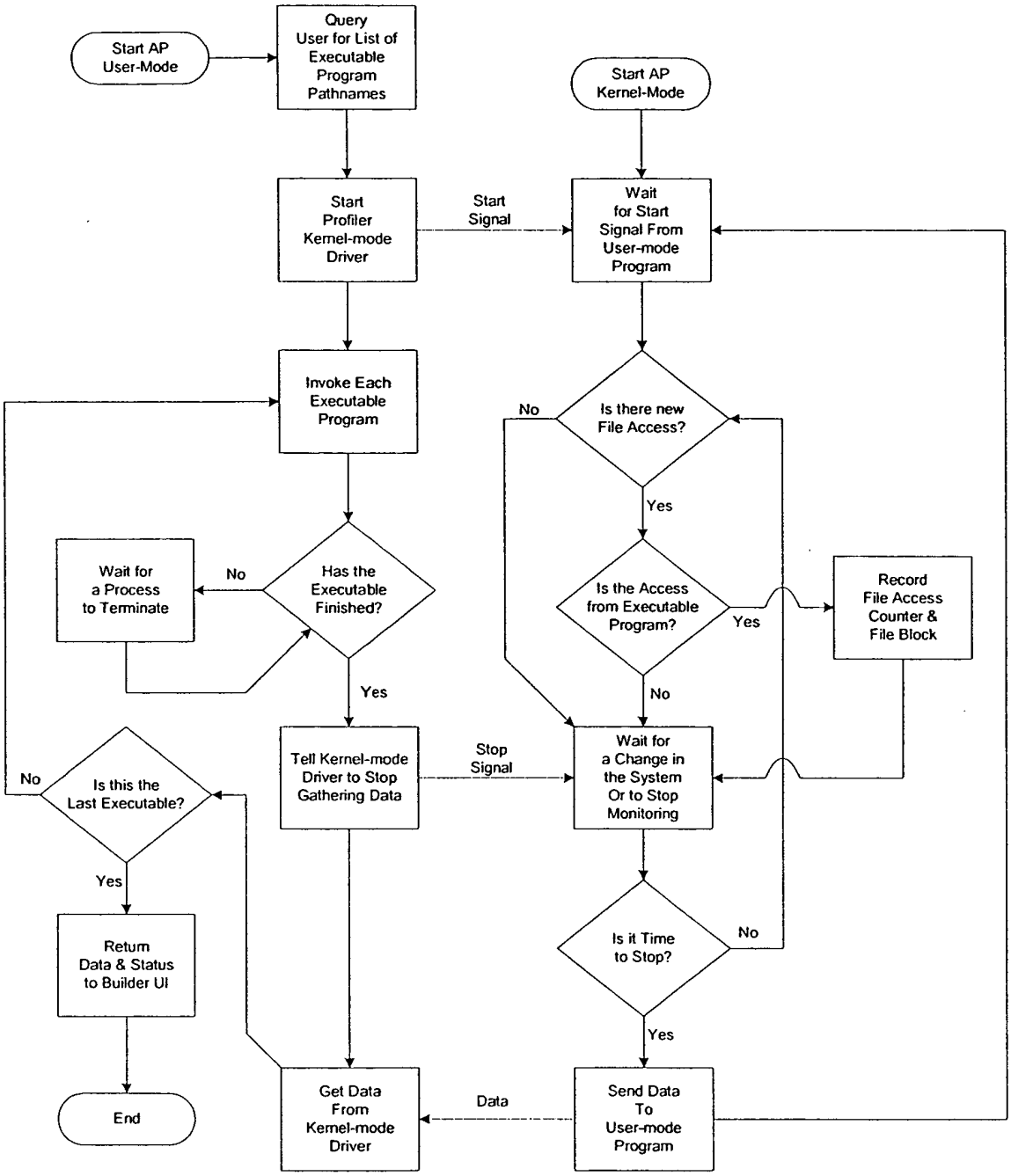
In the second phase of the conversion process, the Builder's Application Profiler (AP) component invokes the application executable program that is installed during the first phase of the conversion process. The executable program files are accessed in a particular sequence. And the purpose of the AP is to captures this sequence data. This data is useful in several ways.

First of all, frequently used file blocks can be streamed to the eStream client before other less used file blocks. A frequently used file blocks is cached locally on the eStream client cache before the user starts using the streamed application for the first time. This has the effect of making the streamed application as responsive to the user as the locally installed application by hiding any long network latency and bandwidth problems.

Secondly, the frequently accessed files can be reordered in the directory to allow faster lookup. This optimization is useful for directories with large number of files. When the eStream client looks up a frequently used file in a directory, it finds this file early in the directory search. In an application run with many directory queries, the potential performance gain is significant.

The Application Profiler (AP) is not as tied to the system as the Installation Monitor (IM) but there is still some OS dependent issue. In the Windows system, the AP still has two subcomponents: kernel-mode (AP-KM) subcomponent and the user-mode (AP-UM) subcomponent. The AP-UM invokes the converting application executable. Then AP-UM starts the AP-KM to track the sequences of file block accesses by the application. Finally when the application exits after the desired amount of sequence data is gathered, the AP-UM retrieves the data from AP-KM and forwards the data to the eStream Packager.

Builder Profiler Control Flow Diagram



EStream Packaging

In the final phase of the conversion process, the Builder's eStream Packager (EP) component processes the data structure from IM and AP to create a data set suitable for streaming over the network. This converted data set is called the eStream Set and is suitable for uploading to the eStream Servers.

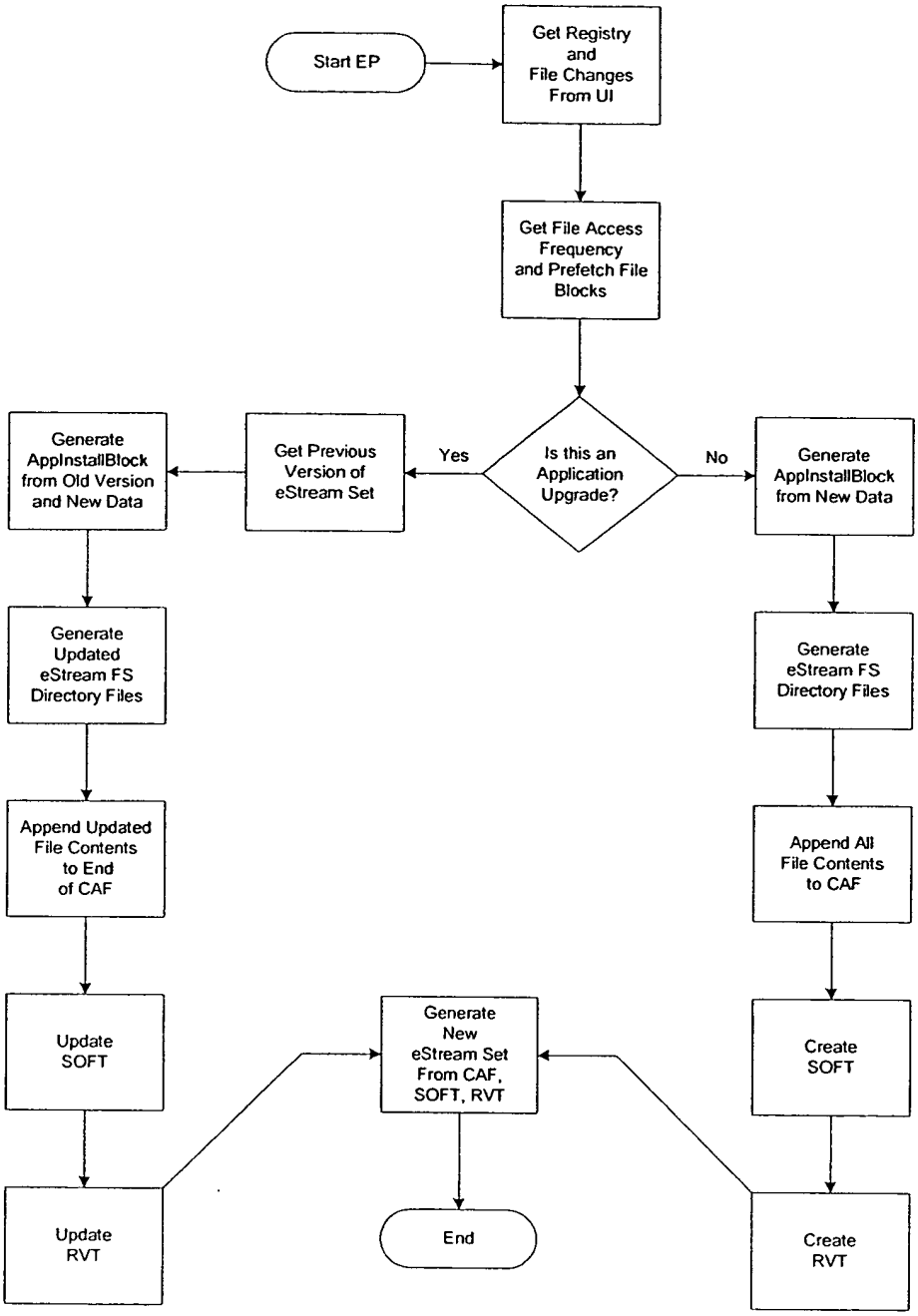
The eStream Set consists of the three sets of data from the eStream Server's perspective. The three types of data are Concatenation Application File (CAF), Size Offset File Table (SOFT), and Root Versioning Table (RVT).

The Concatenation Application File (CAF) consists of all the files and directories needed to stream to the client. The CAF can be further divided into two subsets: initialization data set and the runtime data set. The initialization data set is the first set of data to be streamed from the server to the client. This data set contains the information captured by IM and AP needed by the client to prepare the client machine for eStreaming this particular application. This initialization data set is also called the AppInstallBlock. Detailed format description of the AppInstallBlock is described in another document. The second part of the CAF consists of the runtime data set. This is the rest of the data that is streamed to the client once the client machine is initialized for this particular application. The EP appends every files recorded by IM into the CAF and generates all directories. Each directory contains list of file name, file number, and the metadata associated with the files in that particular directory.

The EP is also responsible for generating the SOFT file. This is a table used to index into the CAF for determining the start and the end of a file. The server uses this information to quickly access the proper file within the directory.

Finally, the EP creates the RVT file. The Root Versioning Table contains a list of root file number and version number. This information is used to track minor application patches and upgrades. The EP generates new directories when any single file is changed from the patch upgrade. The RVT is uploaded to the server and requested by the eStream client at appropriate time for the most updated version of the application by a simple comparison of the client's eStream application root file number with the RVT table located on the server.

Builder eStream Packager Control Flow Diagram



Data Flow Description

The following list describes the data that is passed from one component to another. The numbers corresponds to the numbering in the Data Flow diagram.

1. The full pathname of the installer program is query from the user of the Builder program and is sent to the Install Monitor.

2. The Install Monitor (IM) user-mode sends a read request to the hard-drive controller to spawn a new process for installing the application on the local machine.
3. The OS loads the application installer program into memory and run the installer program.
4. The installer program reads more files from the CD media.
5. The CD media data files are read into memory by the installer program.
6. The application installer program writes the files into proper locations on the local hard-drive.
7. IM kernel-mode captures all file read/write requests and all registry read/write requests by the installer program.
8. IM kernel-mode program sends the list of all file changes and all registry changes to the IM user-mode program.
9. IM user-mode identify special files which needs to be copied or spoofed into eStream client machine before the regular files can be streamed. It also assigns unique file numbers to every file. This data is returned to the Builder UI.
10. Builder UI invokes Application Profiling (AP) user-mode program by querying the user for the list of application executable names to be profiled.
11. Application Profiler user-mode invokes each application executable in succession by spawning each program in a new process.
12. The OS loads the application executable into memory and run the executable.
13. The executable file image is loaded into memory and starts executing. The application files will continuously be loaded into memory as needed.
14. Every file accesses to load the application file blocks into memory is monitored by the Application Profiler (AP) kernel-mode.
15. Application Profiler kernel-mode returns the file access sequence and frequency information to the user-mode program.
16. Application Profiler returns the processed profile information. This has two sections. The first section is used to identify frequency of files accessed. The second section is used to list the file blocks for prefetch to the client.
17. The eStream Packager receives files and registry changes from the Builder UI. It also receives the file access frequency and a list of file blocks from the Profiler.
18. The eStream Packager reads all file data from the hard-drive that are copied there by the application installer.
19. The eStream Packager also reads the previous version of eStream Set for support of minor patch upgrades.
20. Finally, the new eStream Set data is stored back to non-volatile storage.

Mapping of Data Flow to eStream Set

Step 7: Data gathered from this step consist of the registry and file changes. This data is mapped to the AppInstallBlock's File Section, Add Registry Section, and Remove Registry Section.

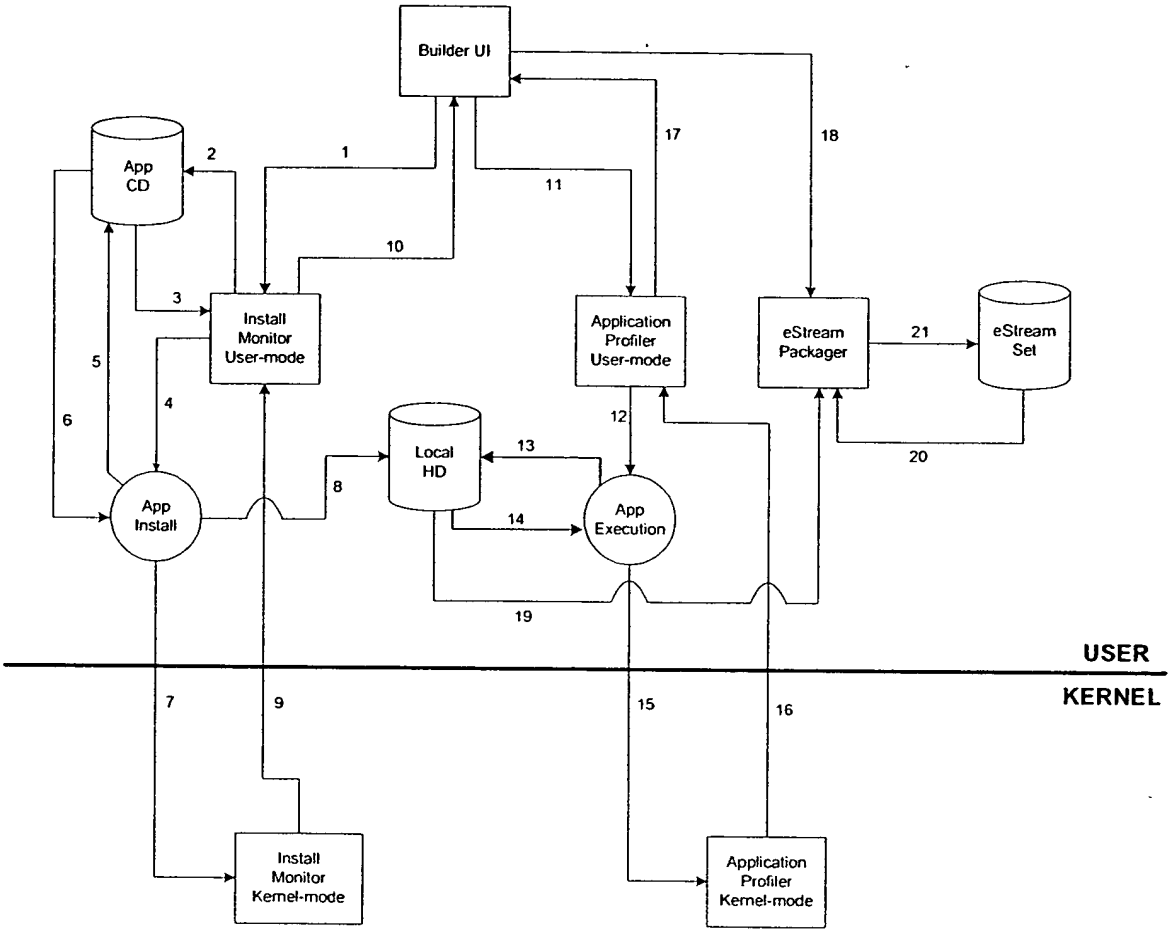
Step 8 & 19: File data are copied to the local hard-drive then concatenated into part of the CAF contents.

Step 10: Data returned to the Builder UI contains unique file numbers. This data is mapped to the file numbers used throughout the eStream Set data structure.

Step 15: Part of the data gathered by the Profiler is used to generate a more efficient eStream FS Directory content. Another part of the data is used in the AppInstallBlock as a prefetch hint to the eStream client.

Step 20: If the installation program was an upgrade, eStream Packager needs previous version of the eStream Set data. Appropriate data from the previous version is combined with the new data to form the new eStream Set.

eStream Builder Data Flow Diagram



v01

Format of eStream Set

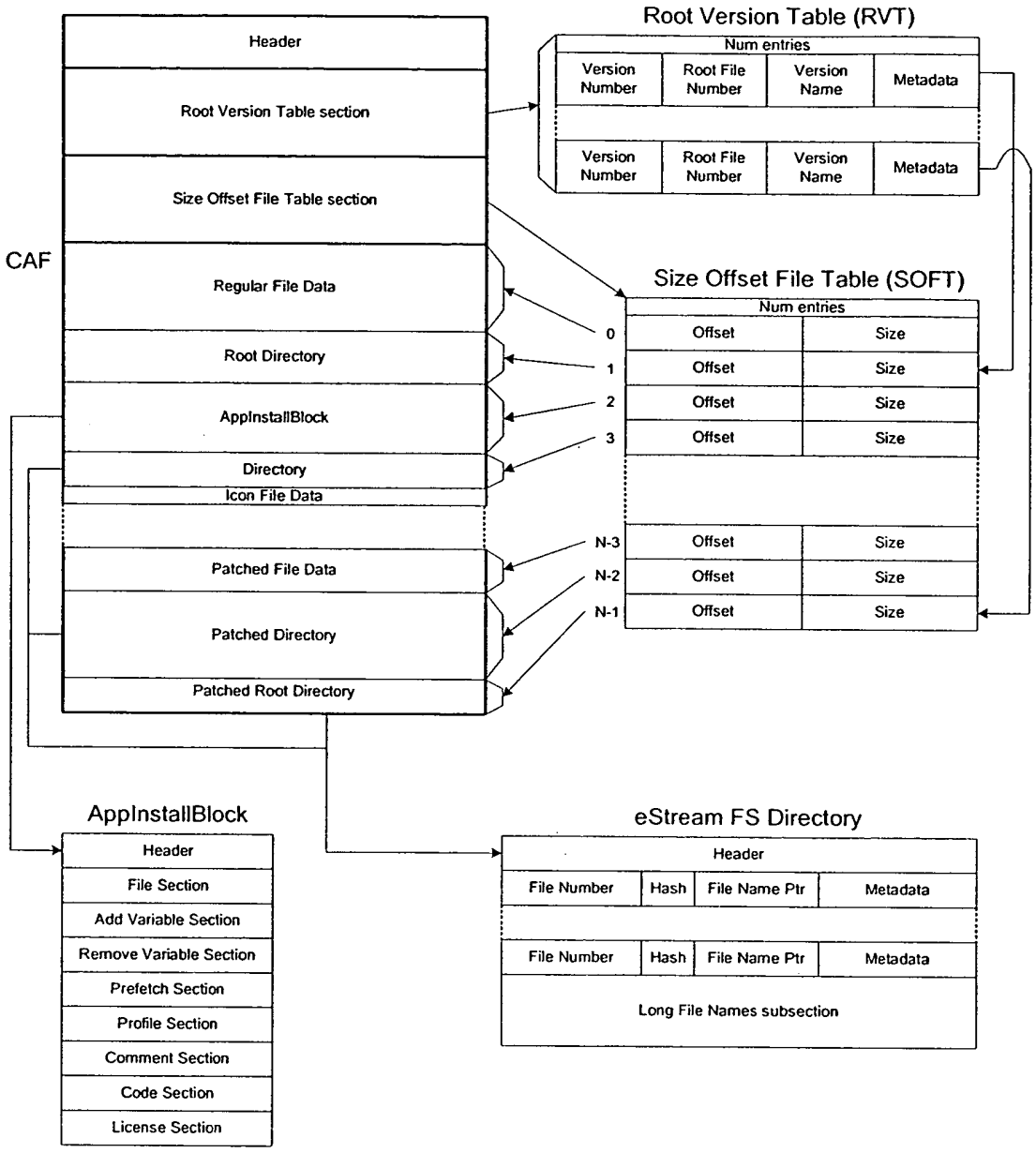
The format of the eStream Set consists of 3 sections: Root Version Table (RVT), Size Offset File Table (SOFT), and Concatenation Application File (CAF). The RVT section lists all versions of the root file numbers available in an eStream Set. The SOFT section consists of the pointers into the CAF section for every file in the CAF. The CAF section contains the concatenation of all the files. The CAF section is made up of regular application files, eStream FS directory files, AppInstallBlock, and icon files. Please see the document on eStream Set Format for detailed format of the eStream Set.

OS dependent format

The format of the eStream Set is designed to be as portable as possible across all OS platforms. At the highest level, the format of CAF, SOFT, and RVT that make up the format of eStream Set are completely portable across any OS platforms. The only critical piece of data structure that is OS dependent is located in the initialization data set called AppInstallBlock in the CAF. This data is dependent on the type of OS due to the differences in low-level system differences among different OS. For example, the Microsoft Windows contain system environment variables called the Registry. The Registry has a particular tree format not found in other operating systems like UNIX or MacOS.

Another OS dependent format is the format of the file names. Applications running on the Windows environment inherit the old MSDOS 8.3 file name format. To support this properly, the format of the Directory file in CAF requires an additional 8.3 field. This field is not needed in other operating systems like UNIX or MacOS.

Format of the eStream Set



v 0.1

Device driver versus file system paradigm

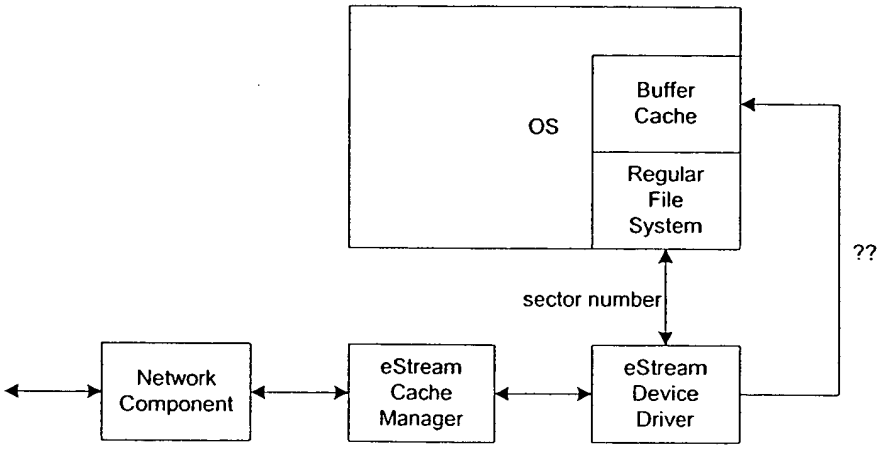
The eStream Prototype is implemented using the 'device driver' paradigm. One of the advantages of the device driver approach is that the caching of the sector blocks is

simpler. The client cache manager only needs to track sector number in its cache. In comparison with the 'file system' paradigm, more complex data structure is required to track a subset of a file that is cached on a client machine. This makes 'device driver' paradigm easier to implement.

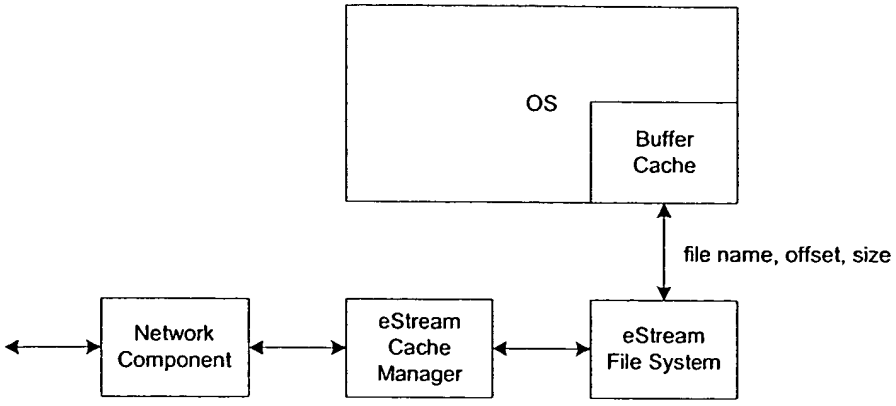
On the other hand, there are many drawbacks to the 'device driver' paradigm. On the Windows system, the device driver approach has problem supporting large number of applications. This is due to the limitation on the number of assignable drive letters available in a Windows system (26 letters); and the fact that each application needs to be located in its own device. Note that having multiple applications in a device is possible, but then the server needs to maintain exponential number of devices that support all possible combinations of applications. This is too costly to maintain on the server.

Another problem with the device driver approach is that the device driver operates at the disk sector level. This is a much lower level than operating at the file level in the file system approach. The device driver does not know anything about files. Thus, the device driver cannot easily interact with the file level issues. For example, spoofing files and interacting with OS buffer cache is nearly impossible with device driver approach. But both spoofing files and interacting with OS buffer cache is need to get higher performance.

Device Driver Paradigm



File System Paradigm



Implementation in the Prototype

The prototype has been implemented and tested successfully on the Windows and Linux distributed system. The prototype is implemented using the 'device driver' paradigm as described above. The exact procedure for streaming application data is described next.

First of all, the prototype server is started on either the Windows or Linux system. The server creates a large local file mimicking large local disk images. Once the disk images are prepared, it listens to TCP/IP ports for any disk sector read or write requests.

Secondly, the conversion process is done on a Windows system via semi-manual procedure. The server disk image is 'mounted' on the local Z drive by making the proper TCP/IP connection to the server. Then the application installation program is invoked and the application is installed into the Z drive. This writes the application files into the Z drive device driver, through the TCP/IP connection, and finally on to the server disk image. At the same time, a file and registry monitoring program records all registry and file changes. This data is stored as an initialization file to be invoked on the client to prepare the client machine for streaming.

Finally, after the application files is stored on the server disk image, the client prototype is started. The client connects to the server and 'mount' the server disk image as a local Z drive. Then the initialization file is invoked which setup the local registry variables and copy system files into proper directories. Once the local machine is prepared for streaming that particular application, the user can start using the application. When the application is first started, the pages are not located in the local buffer cache. The OS makes sector request to the eStream device driver that forwards the sector request to the eStream Cache Manager. If the sector is located in the eStream cache, then the data is returned immediately. If the data is not located in the eStream cache, then the request forwarded to the network component that sends the message to the server. The server finds the proper sector data and returns the data to the client. The client eStream Cache Manager caches the new sector data and forwards the sector data to the eStream device driver. The device driver returns the sector data to the OS.

Exhibit C13

eStream Builder Data Flow Diagram

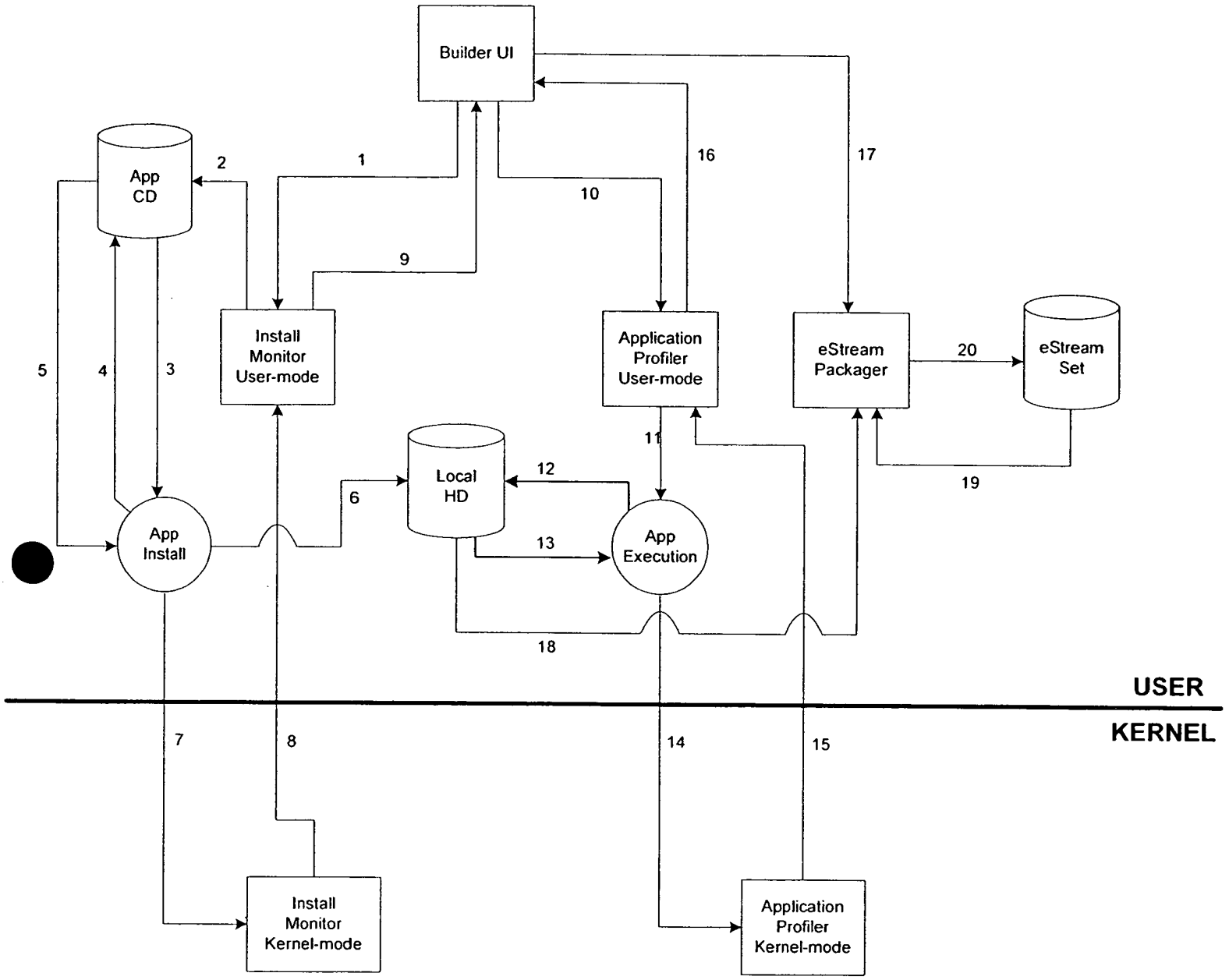


Fig 1

Builder eStream Packager Control Flow Diagram

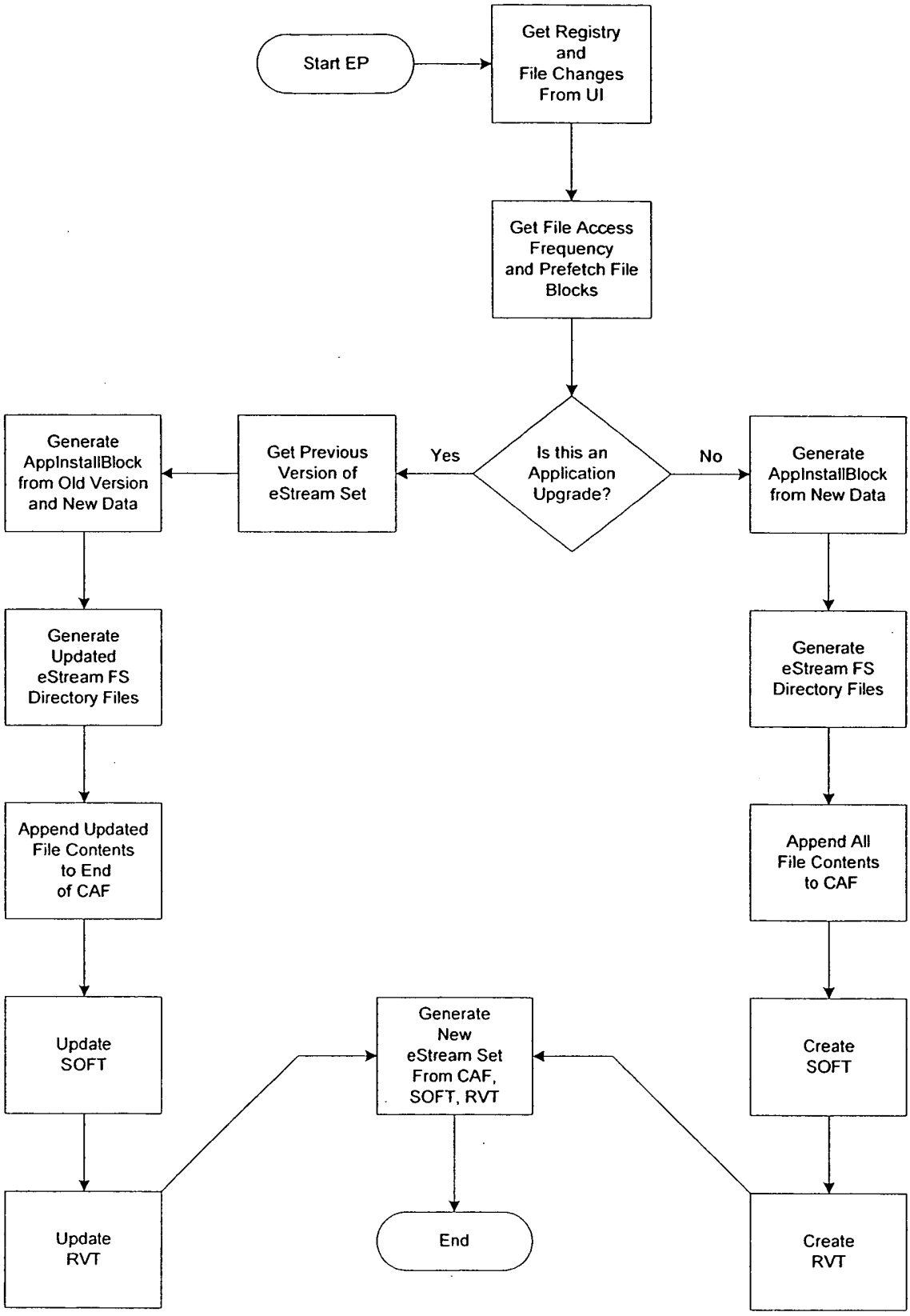


Fig 2

Exhibit C14

eStream Builder Package Manager Low Level Design

Sanjay Pujare and David Lin

Version 0.1

[redacted]

Functionality

The eStream Application Builder Package Manager is responsible for packaging data gathered from the Installation Monitor, the Profile Manager, and the Upgrade Monitor into a set of data called the eStream Set. For the detail format of the eStream Set, see the separate document on eStream Set.

The Package Manager must perform the following task:

- Create the appInstallBlock containing C-File and Registry data from the Install Monitor; Prefetch data from the Profile Manager; and Updated C-File and Updated Registry data from the Upgrade Monitor
- Create a custom installation DLL needed by a specific applications and add to the appInstallBlock
- Create directory files associated with each directory of the application director and add metadata to the directory
- Create directory files associated with each Windows directory containing both the Spoofed files and Z-files
- Create Concatenated Application File (CAF) which is just a juxtaposition of the application files, eStream directory files, and AppInstallBlock
- Create Size Offset File Table (SOFT) which is a mapping of fileNumber to offset of the start of the CAF file
- Create Root Version Table (RVT) which is a mapping from the version of root to the file number of the root directory file
- Archive the CAF, SOFT, and RVT into a single structure called eStream Set suitable for uploading to the eStream Servers.

Data type definitions

The Package Manager doesn't have any internal data types. It must accept and understand data structures received from the Install Monitor and the Profile Manager. See Install Monitor and Profile Manager components for the description of the data structures.

The Install Monitor is responsible for generating the following list of information: list of copied-files, list of spoof-files, list of files with file numbers, list of add registry entries, and list of delete registry entries. The list of copied-files contains the files copied into

non-application specific directories. The list of spoof-files consists of the files too large to be downloaded to the client in the AppInstallBlock. Those files are copied into some special directory on the Z drive for streaming. The list of files with file numbers consists of the files copied into the standard "Program Files" directory and the files that will be spoofed. The registry information is a list of registry key added or removed during the installation of the application.

```
Struct FileIndexTable {
   UINT NumEntries;
   Struct Entry {
      PUNICODE_STRING FilePathName;
      ULONG FileNumber;
   } Entries[NumEntries];
};
Struct FileCopied {
   UINT NumEntries;
   Struct Entry {
      PUNICODE_STRING FilePathName;
   } Entries[NumEntries];
}
Struct FileSpoofed {
   UINT NumEntries;
   Struct Entry {
      PUNICODE_STRING OldFilePathName;
      PUNICODE_STRING NewFilePathName;
   } Entries[NumEntries];
};
Struct RegistryInfo {
   UINT NumEntries;
   Struct Entry {
      PUNICODE_STRING KeyName;
      PUNICODE_STRING ValueName;
      PVALUE_DATA ValueData;
   } Entries[NumEntries];
};
Struct IniInfo {
   UINT NumFiles;
   Struct FileEntry {
      PUNICODE_STRING FilePathName;
      UINT NumSections;
      Struct SectionEntry {
         PUNICODE_STRING SectionName;
         UINT NumValues;
         Struct Entry {
            PUNICODE_STRING ValueName;
            PVALUE_DATA ValueData;
         } Entries[NumValues];
      } Entries[NumSections];
```

```
    } Entries[NumFiles];
};
```

The Profile Manager generates AccessCounts and the PrefetchBlocks data with the structures shown below.

```
Struct AccessCounts {
   UINT NumEntries;
   Struct Entry {
      PUNICODE_STRING FilePathName;
      ULONG Frequency;
   } Entries[NumEntries];
};
Struct PrefetchBlocks {
   UINT NumEntries;
   Struct Entry {
      PUNICODE_STRING FilePathName;
      ULONG BlockNumber;
   } Entries[NumEntries];
};
```

The eStream Set has the following data structure (described in more detail in the separate eStream Set document):

```
Struct eStreamSet {
   Struct eStreamSetHeader header;
   Struct eStreamSetRVT rvt;
   Struct eStreamSetSOFT soft;
   Struct eStreamSetCAF caf;
};
```

Interface definitions

Function 1 : CreateEStreamSet

```
// Create the initial eStream Set from the data
// retrieved from the Install Monitor and the
// Profile Manager.
// This function is called only by the Builder
// UI after data is obtained from Install
// Monitor and Profile Manager.
int CreateEStreamSet(
   IN PFILE_INDEX_TABLE FIT,
   IN PFILE_SPOOFED SpoofFiles,
   IN PFILE_COPIED CopiedFiles,
   IN PREGISTRY_INFO AddRegistry,
   IN PREGISTRY_INFO RemoveRegistry,
   IN PINI_INFO IniInfo,
   IN PACCESS_COUNTS AccessCounts,
   IN PPREFETCH_BLOCKS PrefetchBlocks,
```

```
    IN PVOID DllCode,
    IN PUNICODE_STRING Comment,
    OUT PESTREAM_SET EstreamSet)

Input:
    FIT: File Index Tree contains the file
      number of the directories, spoofed
      files, and standard files

    CopiedFiles: pointer to a list of files
      To be copied to AppInstallBlock

    SpoofFiles: pointer to a list of files
      To be spoofed on the client

    AddRegistry: pointer to a list of registry
      Data to add
    RemoveRegistry: pointer to a list of
      Registry data to remove
    IniInfo: pointer to a list of ini changes

    AccessCounts: pointer to the list of
      Files with the access frequency

    PrefetchBlocks: pointer to the prefetch data
      To be inserted into the appInstallBlock
      Of the eStream Set

    DllCode: pointer to DLL Code

    Comment: pointer to comment string

Output:
    EstreamSet: pointer to the eStream Set

Return Value:
    Success or failure of the packaging process

Comments:
    The eStream Set will be large for most
    application.  Intermediate data will be
    stored on the local hard-drive.

Errors:
    OutOfStorage: failure to find enough storage
      For this eStream Set
```

```
   FileNotFound: failure to find the files
     Specified by either ListCFiles or
     ListZFiles
```

Function 2 : UpgradeEStreamSet

```
// Upgrade the eStream Set to the latest
// version. This function is only called by
// the Upgrade Manager within the same process.

int UpgradeEStreamSet(
   INOUT PESTREAM_SET EstreamSet,
   IN PFILE_INDEX_TABLE UpgFIT,
   IN PFILE_SPOOFED UpgSpoofFiles,
   IN PFILE_COPIED UpgCopiedFiles,
   IN PREGISTRY_INFO UpgAddRegistry,
   IN PREGISTRY_INFO UpgRemoveRegistry,
   IN PACCESS_COUNTS UpgAccessCounts,
   IN PPREFETCH_BLOCKS UpgPrefetchBlocks,
   IN PVOID UpgDllCode,
   IN PUNICODE_STRING UpgComment)

Input:
   UpgFIT: File Index Tree contains the file
     number of the directories, spoofed
     files, and standard files

   UpgCopiedFiles: pointer to a list of files
     To be copied to AppInstallBlock

   UpgSpoofFiles: pointer to a list of files
     To be spoofed on the client

   UpgAddRegistry: pointer to a list of
     Registry data to add

   UpgRemoveRegistry: pointer to a list of
     Registry data to remove

   UpgAccessCounts: pointer to the list of
     Files with the access frequency

   UpgPrefetchBlocks: pointer to the prefetch
     Data to be inserted into the
     AppInstallBlock Of the eStream Set

   UpgDllCode: pointer to DLL Code
```

```
   UpgComment: pointer to comment string

Output:
   EstreamSet: pointer to the eStream Set

Return Value:
   Success or failure of the packaging process

Comments:
   The eStream Set will be large for most
   application.  Intermediate data will be
   stored on the local hard-drive.

Errors:
   OutOfStorage: failure to find enough storage
      For this eStream Set

   FileNotFound: failure to find the files
      Specified by either ListCFiles or
      ListZFiles
```

Function 3 : InsertProfileData

```
// Insert profile and prefetch data into the
// eStream Set. This function is only called by
// the Merge Manager within the same process.

int InsertProfileData(
   INOUT PESTREAM_SET EstreamSet,
   IN PACCESS_COUNTS AccessCounts,
   IN PPREFETCH_BLOCKS PrefetchBlocks)

Input:
   EstreamSet: pointer to old eStream Set
      Before the insertion of the profile
      Data

   AccessCounts: pointer to the list of
      Files with the access frequency

   PrefetchBlocks: pointer to the prefetch data
      To be inserted into the appInstallBlock
      Of the eStream Set

Output:
   EstreamSet: pointer to the new eStream Set

Return Value:
```

```
      Success or failure of the insertion process

   Comments:
      The eStream Set will be large for most
      application.  Intermediate data will be
      stored on the local hard-drive.

   Errors:
      OutOfStorage: failure to find enough storage
         For this eStream Set

      FileNotFound: failure to find the files
         Associated with the prefetch blocks
```

Component design

The pseudo-code for the function *CreateEStreamSet* is described below:

```
{
   Create AppInstallBlock (AIB) from the following input files:
         o SpoofFiles
         o CopiedFiles
         o AddRegistry
         o RemoveRegistry
         o Prefetch
         o Comment
         o DLLcode
   Assign AppInstallBlock with a unique fileNumber given by the IM;
   Record Root fileNumber in the first entry of Root fileNumber Table (RFT);
   Move AppInstallBlock under the Root directory by adding a new entry in the
      Directory structure;
   Create a Concatenation Application File (CAF) header;
   Create a Size Offset File Table (SOFT) header;
   For each (file in FIT) {
      If (file is a directory) {
         Create the directory with new list of fileNumber, filename, and
            Metadata;
      } Else {
         Find the file in the proper location on the HD;
      }
      Append the file or directory to the end of the CAF file;
```

```
        Append the fileNumber, offset into CAF, and size of file in SOFT;
    }
    Archive CAF, SOFT, and RFT into a single eStream Set;
    Return eStream Set;
}
```

The pseudo-code for the function *UpgradeEStreamSet* is mentioned below:

```
{
    Extract previous version PrevAppInstallBlock from eStream Set;
    Create new AppInstallBlock with new FileNumber;

    Extract PrevSpoofFiles and PrevCopiedFiles from PrevAppInstallBlock;
    Divide the C-Files into SpoofFiles and CopiedFiles;
    Add PrevSpoofFiles to SpoofFiles;
    Add PrevCopiedFiles to CopiedFiles;

    Extract PrevAddRegistry and PrevRemoveRegistry data from
        PrevAppInstallBlock;
    Add any unique ((UpgAddRegistry plus PrevAddRegistry) minus
        UpgRemoveRegistry) in the new AppInstallBlock AddRegistry section;
    Add any unique ((UpgRemoveRegistry plus PrevRemoveRegistry) minus
        UpgAddRegistry in the new AppInstallBlock;

    Add UpgPrefetch data to new AppInstallBlock;
    Add UpgDllCode data to new AppInstallBlock;
    Add UpgComment data to new AppInstallBlock;

    For each (directory in UpgFIT) {
        If (any child fileNumber has changed) {
            Create new directory with updated fileNumber;
            Append file to end of Concatination Application File (CAF);
            Append Size Offset File Table (SOFT) with new entry;
        }
    }
    Append new AppInstallBlock to the end of CAF file;

    Prepend Root FileNumber Table (RFT) with new Root entry;
```

Archive CAF, SOFT, and RFT into a single eStream Set;

Return eStream Set;

}

The pseudo-code for the function *InsertProfileData* is mentioned below:

{

// not needed unless merging of uploaded profile data is supported

}

Testing design

This document must have a discussion of how the component is to be tested.

- **Unit testing plans**

  The plan for unit testing Package Manager includes the development of a driver program. This driver interfaces to the Package Manager and invokes the functions with different parameters. The list of possible cases is described below:

  1. Test all interfaces by driving the input parameters with different type of add and remove registry values.
  2. Test all interfaces by driving the input parameters by varying numbers of spoof and copied files.
  3. Test all interfaces by driving the input parameters with some prefetch information.
  4. Test all interfaces for meaningless input values from the IM and PM.
     - Prefetch block containing file number not assigned by IM.
     - IM assigning non-continguous file numbers.
  5. Test upgrade interface for capability to detect and handle bad eStream Set gracefully.
  6. Test upgrade interface and make sure it can detect overlapping file number assignments.
  7. Test upgrade interface and make sure prefetch blocks are not referencing old file number from previous versions.

- **Stress testing plans**

- **Coverage testing plans**

- **Cross-component testing plans**

  The output data from the Package Manager is called the eStream Set. This eStream Set is the input to a stand-alone test program called the *eStream Extrac-*

tor. The Extractor unpacks and 'install' the eStream Set into the local machine without an eStream client file system installed. This test is used to quickly verify that the eStream Set can be run on a pristine machine. Some of the possible variations of the Extractor test includes:

1. Non-default system variable names. I.e. %SystemRoot% set to "D:\Win" instead of "C:\Winnt".
2. Non-default eStream FS drive letter. Use Y instead of Z.

Upgrading/Supportability/Deployment design

The Package Manager logs all error messages to a predefined file common to all components of the Builder program. Every Builder component prints the error message along with its component name. This allows the user of the Builder program to quickly track down any problem during the Building of a new eStream Set.

Open Issues

- Which Builder component creates the installation DLL when the application needs the custom installation code? Is a new component needed to create the custom DLL separately and insert into AppInstallBlock in the eStream Set as needed?

Exhibit C15

eStream Set Format Low Level Design

Sanjay Pujare and David Lin
Version 0.3

Functionality

The eStream Set is a data set associated with an application suitable for streaming over the network. The eStream Set is generated by the eStream Builder program. This program converts locally installable applications into the eStream Set. This document describes the format of the eStream Set.

Note: Fields greater than a single byte is stored in little-endian format. All strings are in Unicode unless specifically stated otherwise. The eStream Set file size is limited to 2^64 bytes.

Data type definitions

The format of the eStream Set consists of 4 sections: header, Root Version Table (RVT), Size Offset File Table (SOFT), and Concatenation Application File (CAF) sections.

1. Header section

- **MagicNumber [4 bytes]:** Magic number identifying the file content with the eStream Set
- **ESSVersion [4 bytes]:** Version number of the eStream Set format.
- **AppID [16 bytes]:** A unique application ID for this application. This field must match the AppID located in the AppInstallBlock. Guidgen is used to create this identifier.

- **RVToffset [8 bytes]:** Byte offset into the start of the RVT section.
- **RVTsize [8 bytes]:** Byte size of the RVT section.
- **SOFToffset [8 bytes]:** Byte offset into the start of the SOFT section.
- **SOFTsize [8 bytes]:** Byte size of the SOFT section.
- **CAFoffset [8 bytes]:** Byte offset into the start of the CAF section.
- **CAFsize [8 bytes]:** Byte size of the CAF section.

- **VendorNameLength [2 bytes]:** Byte length of the vendor name.
- **VendorName [X bytes]:** Name of the software vendor who created this application. I.e. "Microsoft". Null-terminated.
- **AppBaseNameLength [2 bytes]:** Byte length of the application base name.
- **AppBaseName [X bytes]:** Base name of the application. I.e. "Word 2000". Null-terminated.

- **MessageLength [2 bytes]:** Byte length of the message text.
- **Message [X bytes]:** Message text. Null-terminated.

2. Root Version Table (RVT) section

The Root version entries are ordered in a decreasing value according to their file numbers. The Builder generates unique file numbers within each eStream Set in a monotonically increasing value. So larger root file number implies later versions of the same application. The latest root version is located at the top of the section to allow the eStream Server easy access to the data associated with the latest root version.

- **NumberEntries [4 bytes]:** Number of patch versions contained in this eStream Set. The number indicates the number of entries in the Root Version Table (RVT).

Root Version structure: (variable number of entries)

- **VersionNumber [4 bytes]:** Version number of the root directory.
- **FileNumber [4 bytes]:** File number of the root directory.
- **VersionName [32 bytes]:** Application version name. I.e. "SP 1".
- **Metadata [32 bytes]:** See eStream FS Directory for format of the metadata.

3. Size Offset File Table (SOFT) section

The SOFT table contains information to locate specific files in the CAF section. The entries are ordered according to the file number starting from 0 to NumberFiles-1.

- **NumberFiles [4 bytes]:** Number of entries in this section.

SOFT entry structure: (variable number of entries)

- **Offset [8 bytes]:** Byte offset into CAF of the start of this file.
- **Size [8 bytes]:** Byte size of this file. The file is located from address Offset to Offset+Size.

4. Concatenation Application File (CAF) section

CAF is a concatenation of all file or directory data into a single data structure. Each piece of data can be a regular file, an AppInstallBlock, an eStream FS directory file, or an icon file.

a. *Regular Files*

- **FileData [X bytes]:** Content of a regular file

b. *AppInstallBlock (See AppInstallBlock document for detail format)*

A simplified description of the AppInstallBlock is listed here. For exact detail of the individual fields in the AppInstallBlock, please see AppInstallBlock Low-Level Design document.

- **Header section [X bytes]:** Header for AppInstallBlock containing information to identify this AppInstallBlock.
- **Files section [X bytes]:** Section containing file to be copied or spoofed.
- **AddVariable section [X bytes]:** Section containing system variables to be added.
- **RemoveVariable section [X bytes]:** Section containing system variables to be removed.
- **Prefetch section [X bytes]:** Section containing pointers to files to be prefetched to the client.
- **Profile section [X bytes]:** Section containing profile data. (not used in eStream 1.0)
- **Comment section [X bytes]:** Section containing comments about AppInstallBlock.
- **Code section [X bytes]:** Section containing application-specific code needed to prepare local machine for streaming this application
- **LicenseAgreement section [X bytes]:** Section containing licensing agreement message.

c. *EStream Directory*

An eStream Directory contains information about the subdirectories and files located within this directory. The information includes file number, names, and metadata associated with the files.

- **MagicNumber [4 bytes]:** Magic number for eStream directory file.
- **StringTable [4 bytes]:** Byte size offset to beginning of the string table.
- **StringTableLength [4 bytes]:** Byte size length of the string table.
- **ParentFileID [16+4 bytes]:** AppID+FileNumber of the parent directory. AppID is set to 0 if the directory is the root.
- **SelfFileID [16+4 bytes]:** AppID+FileNumber of this directory.
- **NumFiles [4 bytes]:** Number of files in the directory.

Fixed length entry for each file in the directory:

- **FileID [16+4 bytes]:** AppID+FileNumber of each file in this directory.
- **NameHash [4 bytes]:** Hash value of the file name. Algorithm TBD.

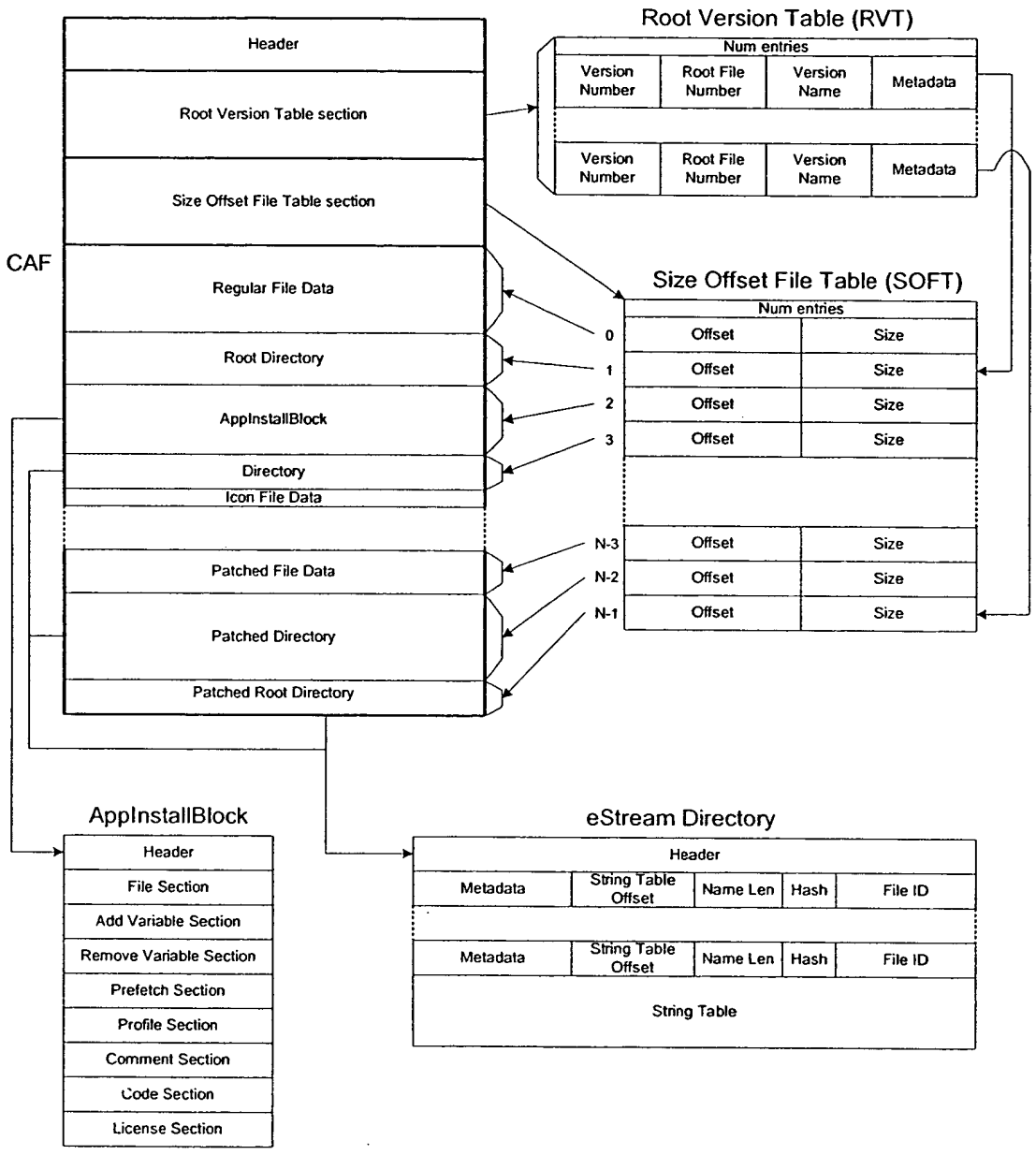
- **FileNameOffset [4 bytes]:** Offset where the file name is located, relative to the beginning of the string table.
- **FileNameLength [4 bytes]:** Byte size length of the file name that is null-terminated.
- **Metadata [32 bytes]:** The metadata consists of file **byte size** (8 bytes), file **creation time** (8 bytes), file **modified time** (8 bytes), **attribute flags** (4 bytes), **eStream flags** (4 bytes). The bits of the **attribute flags** have the following meaning:
  - **Bit 0:** Read-only – Set if file is read-only
  - **Bit 1:** Hidden – Set if file is hidden from user

  The bits of the **eStream flags** have the following meaning:
  - **Bit 0:** ForceUpgrade – Used only on root file. Set if client is forced to upgrade to this particular version if the current root version on the client is older.
  - **Bit 1:** RequireAccessToken – Set if file require access token before client can read it.
  - **Bit 2:** IsDirectory – Set if the file is a eStream Directory.

d. Icon files

- **IconFileData [X bytes]:** Content of an icon file.

Format of the eStream Set



v 0.2

Open Issues

- Where is the metadata associated with the Root directory located? Currently, root metadata is located in the root version table. All other files and directory metadata can be found in their parent directory.

Exhibit C16

eStream App Server Low Level Design

Version 1.2

Sameer Panwar
[redacted]

Functionality

First, some definitions:

eStream page: the smallest unit of data that can be requested by a client from an App Server. Proposed to be 4kB for eStream 1.0.

page set: simply, a sorted list of eStream pages, each identified by a File ID (i.e. AppID & File #) and page # (essentially an offset into the file). This set is restricted only in that all pages in the set must have the same AppID.

client request: a single self-contained message from a client requesting a page set from the server. Each server response to a client request can return a number of pages, and there is a maximum number of pages that the client can request in this message. (TBD, somewhere between 8 and 20 or so).

The primary job of the App Server is to service client requests for application data blocks. The App Server is designed to minimize the amount of CPU time it must consume to satisfy each client request, thereby maximizing scalability. Thus, authentication is performed by a simple expiration time check of an AccessToken provided by the client, and compressed application data is saved persistently.

The App Server serves data derived from eStream Sets. To decouple the performance needs of the App Server from the Builder, we should have a post-processing tool that converts the flat, uncompressed eStream Sets as provided by the Builder into a precompressed format suitable for memory mapping, if the App Server is configured to serve compressed bits. Also, a profiling part of the App Server can be used to monitor for common page sets, and then assemble more optimized replies, which compress the set of pages together as a unit, to take advantage of improved compression ratios. These replies can be stored on disk to save time in rebuilding them each time the server is started up.

The App Server (AS henceforth) views an eStream Set as simply a set of files, and knows no further underlying structure. Thus an eStream Set contains at the start a table (FOST) indexed by File #, and providing the offset into the eStream Set where the associated file data begins, and the size of the file. So the AS just takes the client request of (AppID, File #, Page #, no. of pages), maps AppID to an eStream Set and looks up in the FOST table (File/Offset/Size Table) to find the requested data.

This works slightly differently when the eStream Set file has been pre-compressed by the post-processing tool. The resulting image is the same as before, except now the FOST points to another table, the POST (Page/Offset/Size Table). Because the compressed pages will be of different sizes, this table must by indexed by the Page # to find the relative offset and size of the compressed page data for the file. Thus if an AS is not configured for data compression, the main difference in behavior is that it doesn't do a POST lookup and it doesn't care about coalescing page sequences.

Data type & Data structure definitions

Processed eStream set – this structure is kept on disk and never changes after installation. It looks like:

```
struct {
     ApplicationID appID;     /* for reference, is a 128-bit GUID, see ECM
LLD */
     uint32 maxFileNo;
     boolean compressed_flag;   /* indicates whether the AppFiles are com-
pressed, though maybe we should do it differently? */
     FOST_Entry FOST[<maxFileNo>];
     uint8 appData[<sum of all AppFile sizes, which are variable>] ;
} ProcessedEstreamSet;
```

Since the files in the application are of variable size, we can't make a table out of them, and must indirect out of a table (indexed by the File #) to find their offset location inside the AppData buffer.

```
struct {
     uint32 offset;
     uint32 size;
} FOST_Entry;
```

When the processed eStream set is compressed, then we use the AppFileCompressed structure at the offset indicated by the FOST, otherwise we interpret the data as just AppFile. The AppFileCompressed structure starts with a table that indicates the size and offset of the compressed data that belong to the page it was indexed by.

```
struct {
     uint8 fileData[<size from FOST_entry>]
} AppFile;

struct {
     POST_Entry POST[<number of pages, derived from size from FOST_Entry>] ;
     uint8 fileData[<sum of all FilePage sizes, which are variable>] ;
} AppFileCompressed;

struct {
     uint32 offset;
```

```
    uint32 size;
} POST_Entry;
```

This covers all the structures that live on disk. When we mmap-per-file, that means we make multiple mappings out of a single ProcessedEstreamSet file, at different offsets, one for each file.

Now, for the in-memory data structures (assuming per-file-mmapping):

The primary lookup will be a hash table, hashed on the AppID and FileNo. It should have on the order of 10,000 entries, each table entry containing a list of entries (for collisions). Each list entry contains:

```
struct {
    ApplicationID appID;
    uint32 fileNo;
    uint32 size;   /* size of the mapped Appfile */
    MMap fileMap;
    HTListEntry * next;
} HTListEntry;
```

The Mmap struct just contains any OS-specific-related stuff to manage the mappings, plus a field `char * ptr`, which points to the place in memory that the AppFile (or AppFileCompressed) is mapped. So the hash table looks like:

```
struct {
    HTListEntry * entry[<size of hash table>];
} MMapHT;
```

Hash function is TBD. The hash table should be statically sized large enough to handle the full number of eStream sets up to the maximum memory we will support. Assuming 32 bytes being used per entry, that implies about 1 MB to handle 30k files, which is no problem. (Maybe we should reserve entries for 100k files or more?)

Configuration: Each AS must obtain configuration data, either directly from the database or from the monitor in its startup message. The required data is (with the config param names and datatypes):

```
AppList         vector of ApplicationID's (128-bit GUIDs)
ServerPort      uint16
MonitorPort     uint16
SLiMKey         uint (size TBD, depends on actual algorithm)
ClientTimeOut   uint32
CompressionFlag uint32
```

Network communication: The AS talks only to clients and the server monitor via the network. The server monitor communication will be described as part of the monitor heartbeat protocol. The AS-client communication will be described in a separate docu-

ment. The AS will time-out and close connections that have been idle for some amount of time (a few seconds).

[maybe combine multiple responses into a single send socket call (will only work for TCP probably, since proxies won't like multiple server responses)?]

Interface definitions

The AS is optimized to do one thing only: serve pages from the read-only file system part of eStream, so there is just one interface with the client. Anything the client can care about in an eStream set is just another file to the AS, including the AppInstallBlock, and directories/metadata. The AS only returns the data the client requested, nothing extra.

```
struct {
     uint32 fileNo;
     uint32 pageNo;
} PageRequest;

struct {
     uint32 errorCode;
     uint32 compressedFlag;
     uint32 fileNo;
     uint32 pageNo;
     uint32 offset; /* offset into pageData below */
     uint32 dataSize;
} PageReply;
```

PageReadRequest

Caller: Client
Callee: AppServer

Input:	uint32	appId;
	eStreamAccessToken	accessToken;
	uint32	numPagesRequested;
	PageRequest	pageSet[(numPagesRequested)];
Output:	uint32	numPagesRequested;
	PageReply	pageSetReply[(numPagesRequested)];
	uint8	pageData[(sum of all page data)];
	uint32	globalErrorCode;

Global Errors: INVALID_ACCESS_TOKEN
EXPIRED_ACCESS_TOKEN

INVALID_APP_ID
EXCEEDED_MAX_REQUESTABLE_PAGES

Errors within PageReply: INVALID_FILE_NO
INVALID_PAGE_NO
SERVER_ERROR (probably should be logged, and should cause an alert if too many occur in some time period, including errors that don't get returned to the client.)

AppServers don't ever talk to the database (it would be a waste of licenses considering the number of AppServers we'd have and their infrequent accesses). Instead, they obtain all their relevant control information from the server monitor.

The exact interfaces are TBD, but from the monitor they will provide configuration information, AppServer state change requests, and add/remove requests to the list of apps being served. Going back from the AppServer to the monitor, it will report load (average response time) on a per app basis, and server state, along with the heartbeat.

Component design

Interesting issues to deal with:

Scalability/Performance

Since scalability (and thus performance) is critical for the AS, let's go over how CPU and memory are used.

Memory

Performance is maximized when virtually all client requests can be satisfied by retrieving the desired pages from RAM, because RAM is far faster than disk. Thus the amount of RAM available will put an upper bound on the number of apps that a single AS can serve efficiently. Since server RAM won't grow as fast as the total size of all apps available as eStream sets, this means we'll have to heterogenize servers, where each server specializes in a subset of apps, limited by available RAM. For eStream 1.0, this component of AS configuration will be handled manually, the eStream administrator assigning apps to servers. In the future, the set of App Servers should automatically reassign apps dynamically to balance load.

But this is just one level of memory, committing RAM to a set of apps. There still remains the question of how to best utilize that RAM for each app, since some files are used far more often than others. This immediately means that for efficiency we must overcommit RAM, because if we allocate an entire eStream set into RAM, we're using precious resource to hold data that may be requested only very rarely. Instead of having to manage our physical RAM manually to accomplish this (such as with a cache), an easier approach would be to take advantage of virtual memory (VM) to automatically keep

the hot pages in RAM, with the remainder available (again **automatically)** off disk (via memory mapping the eStream sets). That way the server can satisfy any possible client request for any app it serves, but is optimized to be the most efficient over all clients. But this only works if enough VM is available. (Time for some back-of-the-envelope numbers.) Given that an app seems to have something like only 20% of it being hot (from our current limited data from the prototype), this means VM must be at least 5x of RAM for maximum efficiency. Given that a process has about 2 GB addressable VM, this corresponds to about 400 MB of RAM. Beyond that size (which is not uncommon), we don't have enough VM to efficiently overcommit our memory (by mmaping entire eStream sets). So now our choice is to either manually manage a memory cache (and all the attendant coding, bugs, etc.), or to mmap at a finer granularity.

Note that the effective virtual memory required by an app is increased when compression is used, to handle the extra compressed page sets. They'll probably double or triple the RAM footprint by hot pages (due to redundancy), but only increase the overall VM footprint by 1.2 – 1.5. The consequence of this is that the overcommit ratio goes down to 1.5 / (3 * .2) = 2.5, though the amount of apps servable is reduced to 1/3 (!!). Now 2 GB virtual address space corresponds to 800 MB of RAM. This means we should be able to just memory map entire eStream sets, up to 2 GB worth, and be confident we're utilizing RAM efficiently, assuming the server has about 800 MB of RAM. A server with less RAM will likely thrash, and those with more will likely see little improvement in the number of apps they can serve via memory mapping.

A loss of 2/3 in the number of apps an AS can serve I think is too great a sacrifice, too great a loss in app scalability (need 3x the number of servers as before!) for what is about a 15-30% greater effective bandwidth at the client. The root of this problem is the redundancy (costly in physical memory), because the compressed page sets will contain the same page in multiple sets. This is similar to the redundancy that appears in trace processors and dynamic translation, which places extra memory demands in both those cases. I think we must completely eliminate this redundancy to achieve the goals we desire, either by (1) not using compressed page sets, and just sending multiple individually compressed pages, or (2) ensuring a page appears in only one compressed page set. [There further potential loss of effective memory size when using compressed page sets since they'll be allocated in 4k chunks, thus wasting about 2k on average; we'd have to batch them up together in files to minimize this... Also, saving the compressed page sets to disk introduces extra complexity to the AS because we'd have to properly handle recovery (i.e. what if the system crashes while we're writing the sets, which if we're memory mapping is totally out of our control). Because of this robustness requirement, and the fact we need to be 100% sure we're serving good bits (lest we crash a bunch of clients), this needs to be thought out very carefully if we want to do this. My opinion is that we should defer implementing compressed page sets until we better understand the tradeoffs, and good profiling schemes. In particular, will the AS be mostly bandwidth-limited, memory-limited or CPU-limited?]

Separately from this, we should consider the effect of per-file memory mapping (ignore the compressed page sets now). This has the impact of requiring many more mmap's

from the OS, but promises better use of the limited virtual address space. In this scheme, we mmap each file into VM as it is referenced by a client. If only hot files are referenced, then the RAM footprint is the same as before, but VM is only used for the hot files, not the entire app, probably about 30-50% greater in size. Thus the overcommit ratio then becomes 1.5, much better than the 5 with full app mmapping. So 2 GB of VM corresponds to 1.3 GB of RAM, much better than the 400 MB with full app mmapping. However, this assumes that VM is used in a cache-like manner, evicting not recently used mmap's, since as uncommon files are referenced, they eat up more and more VM. Once VM is totally used up, then replacement policies and eviction (and fragmentation of virtual address space) become issues, just as with a manually managed cache. One solution is to simply purge all mmap's and start from scratch, which is simple and reliable, especially considering the AS is multithreaded (if this is done, the above analysis doesn't hold, and performance becomes a function of how often VM is cleared). Another possibility might be to use the profiling mechanism and only place sufficiently popular files in mmaps and do regular file system accesses for the rest.

Of course, the alternate option for managing physical memory is to know its size, and manage a cache manually. One advantage here is that the AS would know the physical memory consumption and usage (unlike when the OS was handling everything), which may help with load balancing. The main advantage is that there are no artificial limits (overcommit ratio is irrelevant), and only physical memory size is the true limit, and this approach can map any number of eStream sets (with any size of files) to any amount of physical memory up to the virtual address size (4 GB). Then memory management becomes an issue (what do you do once all your RAM is full), which can be painful in a multithreaded environment. Again, we can just invalidate the whole cache as an option, but this will probably happen more often than with the per-file-mmapping case, unless RAM is greater than the maximum that the per-file-mmaping approach can handle. If the wholesale cleanup approach is used, then allocating fixed size chunks may not be needed, and we could potentially get better memory usage by packing compressed pages more tightly (e.g. 16-byte aligned vs. 4kB aligned), which is another potential advantage. Maybe instead of wholesale cleanup, we mark the most commonly used pages, and then just compact those and dump the rest (say 50%). The main issue with this approach is potential redundancy with respect to the OS disk cache (which is shared in the mmap approach), and assumption that our caching policies will be better than the OS's. Also, lookups get messier, since we need a bigger lookup table to index via page # as well.

Yet another option is to use multiple processes instead of multiple threads, one process per app being served, thereby releasing us from the 2 GB VM limitation. However, this introduces the issue of multiplexing requests from the network via IPC, and more load on the server monitor. On x86 NT, a Very Large Memory feature is available that can provide 36-bit addressing per process; we may want to use this even though it won't be available on regular Unixes (and probably not x86-linux).

In summary: per-eStream-set-mmapping is probably too wasteful of virtual address space. Per-file-mmapping is much better, but then memory management becomes an issue, suggesting a simple throw-away-and-start-over solution. However, given that solu-

tion, if a lot of physical memory is desired, a manual cache approach may be better (the better packing should overcome any loss due to redundancy with the OS disk cache). Compression of page sets invokes several issues that probably can't be fully addressed until after 1.0. **Bottom line: the target now for 1.0 is to use per-file-mmapping with per-page compression (but no compression of page sets).** Also, we should instrument the system to allow us to easily collect the relevant data (mem usage, CPU load of different routines, etc.) to help guide us in further evolution of the system to improve performance (e.g. compressed page sets or explicit page cache).

CPU

The main work of the CPU is as follows (encryption is assumed to be done by hardware since its CPU impact is severe):

1. OS system call to retrieve request from network.
2. Decode client request.
3. Validate AccessToken.
4. Lookup AppID, File # in primary lookup hash table. (If mmapping eStream sets, instead lookup in App table, then lookup in FOST).
5. If mmapping then (if uncompressed, no further lookup, if compressed, then lookup in POST to find page and size), if explicit cache then look in B-tree (secondary lookup).
6. If lookup fails, then bring in the data off disk (either mmap or file system call).
7. Copy page data to reply buffer.
8. OS system call to send reply to network.

However, if compressed page sets are used, lookups get more complicated, with a different set of tables to check for an appropriate page set first (and lookup failures incur potential decompression/compression). It appears the least amount of CPU time is probably incurred when doing per-file-mmapping. All pages held in memory are kept in compressed form to save repeated compression of the same data, so pretty much all the work is in lookups and memory copies. Potentially the AccessToken validation will use hardware assist. Lookup failures (i.e. having to go to disk) should be relatively uncommon, and memory should be sized to ensure that.

However, since the AS will run in user mode, this incurs the penalty of two extra copies (from the network buffers) and switching between kernel and user mode twice. If this is enough of a problem, we'll have to consider implementing the AS to run in the kernel (all commercial NFS, etc. implementations run in the kernel), which means we should choose our implementation to be compatible with that approach. In particular, we may not be able to rely on the virtual address space not being fragmented, so mmapping full eStream sets may be impossible. Plus robustness of the server becomes even more important, and portability issues arise. For the 1.0 release, we plan to implement the AS in user mode keeping the possibility of moving to kernel mode in the future, and will collect data from 1.0 (or derived prototype) to evaluate the actual benefits.

Disk: Since we are relying heavily on the common pages being in memory, we could possibly even consider storing the processed sets on a network disk, i.e. remote from the app server itself. However, such sharing won't work well for compressed page sets since

they are written to at runtime—it would be extremely messy to handle dozens of app servers trying to add many compressed page sets (possibly the same) to a set of shared files.

Multithreading model

The approach will be to have a single boss thread which pulls things out of the network port and stuffs client requests into a queue and a bunch of worker threads which grab requests and send back the replies. Simple enough, but this raises the issue of thread control, since the boss also needs to be able to handle threads that die or hang and kill and restart them. The boss thread will monitor the worker threads and provide load/hearbeat info to the monitor through the server manager thread, thus giving visibility to the server monitor of the health of all the worker threads.

Load balancing

To be described elsewhere? (appears in SLiM server LLD)

Security

There are two levels of security involved in the AS. First, we must prevent clients who don't hold valid licenses from gaining access to the licensed binaries. This is accomplished by the client obtaining an AccessToken from the SLiM server and presenting it to the AS upon every request. The AS can then use the SLiM server's public key to test the authenticity of the AccessToken (to protect against forgeries), and then can test the authentic expiration time of the AccessToken. Second, we must encrypt the actual data being sent on the wire to prevent third parties from gathering the binary data covered by the license. Since the data coming out is somewhat obfuscated anyway (files are identified by arbitrary IDs, with our own strange message formats and compression and all in random pieces, etc.) it is not clear how much extra protection is really necessary, i.e. what do the license issuers actually want? We should use a common scheme like SSL to perform this encryption. It has been decided that the encryption load for this would be too great, and thus the data send back will be unencrypted. We may use SSL for authentication purposes only (i.e. null-cipher), if that is cheap enough.

Also, a possible optimization for checking AccessTokens would be to cache recently used AccessTokens along with a signature/hash. If a token presented by a client matches, then we can skip the authentication step (since we've done it once already) and just check the expiration time.

Robustness

The AS must be very robust. It must catch OS call errors and handle/log them as appropriate, and deal with threads that hang or die. Thus it needs to aggressively check for error conditions and possible failure modes. The AS also needs to track relevant resources (e.g. sockets, memory) and carefully manage/reclaim them so as not to exceed any limits or to degrade performance. And of course, the AS needs to check all data coming in from the client, to deal with ill-formed requests, and illegal values (e.g. huge negative indexes, etc.), and perform no potentially dangerous operation without validating parameters. This becomes even more important when we eventually move the AS to run in kernel mode. The AS also needs to be as stateless as possible, to minimize recovery time, and if it does perform writes to disk (such as for the compressed page sets), do so in a reliable fashion conducive to quick recovery. Any unreliability in the AppServer will negate any benefit of scalability we have over our competitors.

Testing design

This document must have a discussion of how the component is to be tested. Some subsections could include:

Unit testing plans

The various components of the AS are not too large or complicated: The request dispatcher (to worker threads), the hash table, the compression code, the AccessToken checking code, etc. These shouldn't be too hard to do reasonable testing on in isolation.

For the post-processor component, we'll have to build some sample Estream Sets as input, but it'll be hard to tell whether the output is correct without having a minimal working AS.

Cross-component testing plans

The best approach will be to perform incremental implementation and testing. I.e. we build the core functionality that is required (i.e. can start with just regular i/o reads), and then add the more performance-related stuff later (adding mmaping, and then the hash table & AccessToken checks), while testing the entire system as pieces are gradually added (of course performing sanity-check and other minimal testing on the pieces first if possible). Compression can be added last.

To actually drive the AS, we'll need a test client, which will be designed to just shoot off a series of read requests to the server. The file data returned could then be written to files, and this can be compared against the original set of files used to create the Estream Set we started with, to check that the data was received properly. For checking error conditions, a log of errors can be written and compared against a reference log for those requests we expect to fail.

Stress testing plans

To accomplish this, we should run multiple independent test clients (on the same machine and on different machines), and increase the frequency of requests (to stress the AS's threads and sychronization, and communication routines), and the number and size of files referenced (to stress the hash table and memory). Each test client can then check whether the data and errors it got back were as expected, like in the above subsection.

Coverage testing plans

Should we use some kind of code coverage tool for this?

Performance testing plans

Since performance is critical, we should take the time to evaluate the AS's performance characteristics. We need to crank up our stress testing until either bandwidth or CPU saturates, and record the request rate that generated it. We should compare how this point responds to high numbers of clients with fewer requests per client vs. fewer clients with higher requests per client. We'll need to profile the system to find bottlenecks to tweak more performance out of it, and learn how well our original design assumptions hold up. Depending on whether CPU or bandwidth (or memory) saturates first, we may want to modify the system's tradeoffs to improve scalability further, and otherwise note which components a customer should upgrade for better performance. Also, if we think we can come up with reasonable client access pattern profiles, we may want to use those to estimate the actual number of real-world clients an AS can support. As part of this, we'll probably want to run the AS in-house once it is mature enough (eat our own dogfood), and then farm out app upgrades, etc. (play out some of our scenarios) and see what happens to the AS's (do they choke or what).

Availability testing plans

We will also need to test our failover and load balancing capabilities. This will require several test machines with the monitor in place to start and stop servers, and have clients be aware of multiple AS's and respond appropriately when an AS stops responding. For load balancing, we'll probably want a bunch of test clients with a variety of access patterns and see how well their requests are distributed.

Upgrading/Supportability/Deployment design

App Servers will possibly need to version their interface with clients (requiring clients to state the version they're expecting), but will also need to support older versions.. We may also modify the Estream Set format (or just the processed set format), but that should be handled by upgrading both the AS and post processor and then regenerating the processed sets.

For supportability & deployment, the AS will report error conditions and load to the server monitor, which is used by the customer.

Open Issues

1. Is there a limit to the # of possible mmaps?
2. Is there a single system call to unmap all mmaps?

Exhibit C17

eStream 1.0 CORBA Centric Server Framework

Authors: Amit Patel, Bhaven Avalani, Michael Beckman

[redacted]

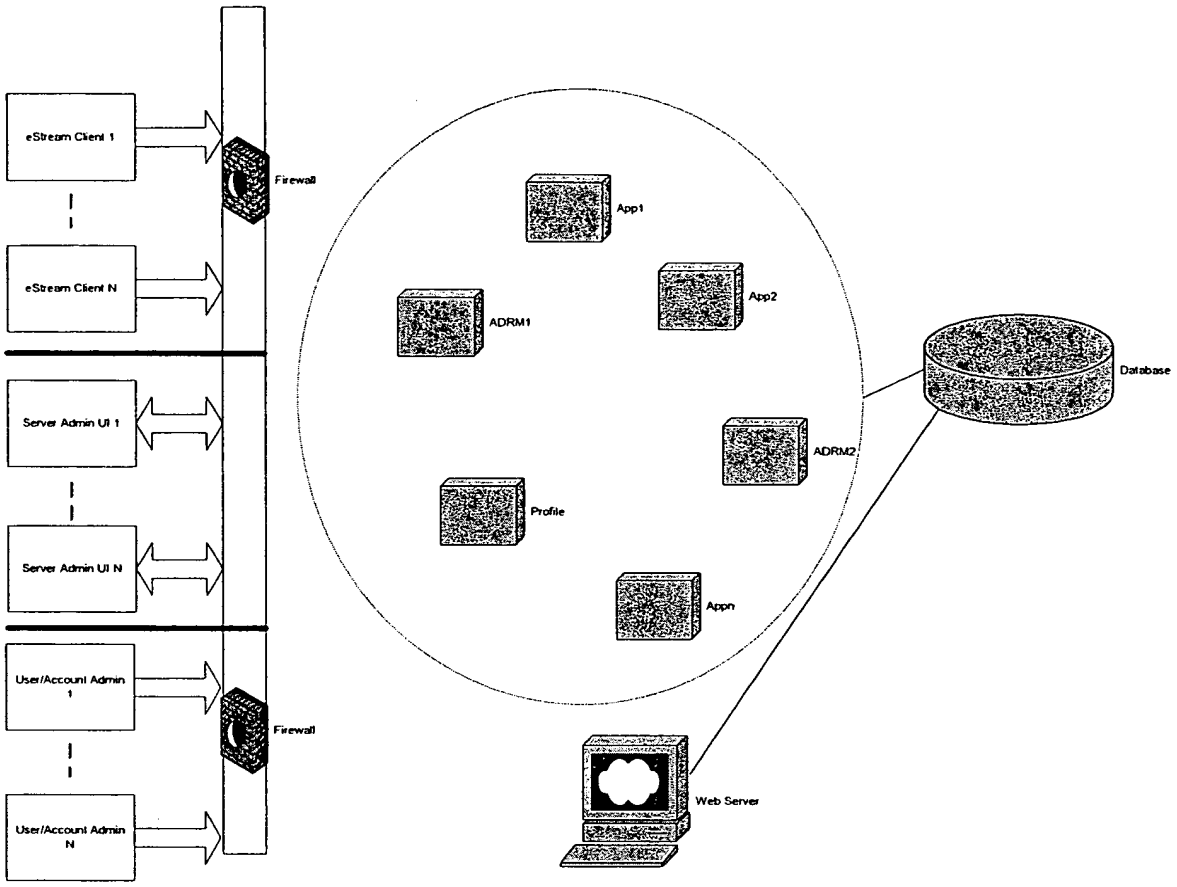
Omnishift Confidential

Abstract: The following document presents a server framework based on CORBA for eStream 1.0.

Descriptions: eStream 1.0 is a distributed server environment. CORBA provides a cross-platform cross-language distributed system solution. The high level essential features of the CORBA framework are listed below.

- Messaging Support. How do the servers and clients, servers and servers talk with each other. CORBA provides mechanism for inter-object communication on a variety of protocols (IIOP, GIOP, IIOP over HTTP).
- Distributed Object Management. This essentially is useful for management and monitoring in eStream as the client side objects eStream supports are fairly simple. However for management and monitoring all servers need to provide objects which advertise the health of the system.
- Services: Corba provides a variety of services for a distributed system.
    - Naming Service. Helps maintain the location of objects in the systems. This is very useful for server management tools.
    - Event Service. Useful for Alarms etc.
    - ORB service. Used for server configuration, server state. It has the capability to stop/start servers.
    - Security service. Useful to access control and encryption services.
    - Distributed Transaction Support. Probably not relevant to our framework.

The following diagram illustrates the eStream architecture at a very coarse level.

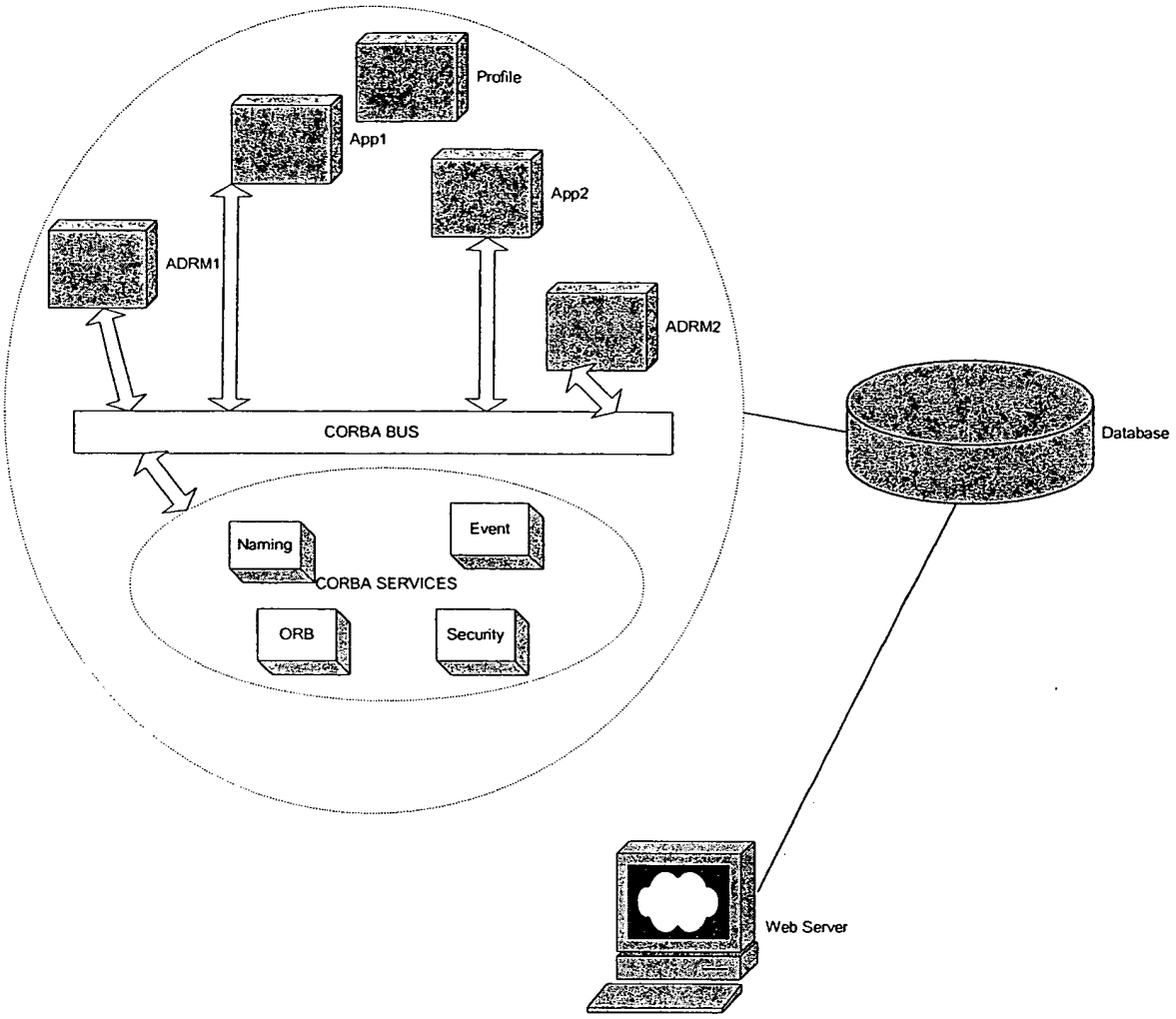


Listed below are the objects in our system and the data they manipulate.

CLIENT	DATA	LOCATION
eStream Client	Account/User/Subscription Information	RDB/LDAP
	EStream Sets	File System/RDB
	Server Information(Location of ADRM, APP, Profile etc)	???
User/Account Management Client	Account/User/Subscription Information	RDB/LDAP
	Server Information(Location of ADRM, APP, Profile etc)	???
Server Administrator Client	Server Information(Location of ADRM, APP, Profile etc)	???
	Real time/Heart Beat	???

	status of the servers	
	Load information for the servers	???
	Configuration information for servers	???
Servers (as Clients)	Static Configuration of other servers in the system. Give me a server which serves Word.	???
	Dynamic Configuration of other servers in the system. Heartbeat and the load are examples of this information.	???
	Load/Logging/Alarm Information. Log this access. Write down my load. Raise this alarm to the administrator.	???

The question marks in the table above are transient data which characterizes the current state of the servers in the system. A CORBA based system will solve this problem using the following server architecture.



The management client in this scenario will essentially talk to the CORBA system to get any information of the servers in our system.

Listed below are the pros and the cons for a CORBA based system.

PROS:

1. A well-defined and proven server framework.
2. Cross platform support.
3. Lot of services is available for free. Alarm, Management, Load Balancing.
4. Distributed. The objects in the system are inherently distributed and hence more scalable.
5. High performance system. Transient data about the system is stored in transient storage and hence data accesses are fast.

6. Tools and services are available for free. Example: A distributed transaction support system is available and may be useful for eStream in the future.
7. Server management and Alarm tools are easily available.

CONS:

1. Vendor Lock in. Visigenix and Iona are primary vendors with their own set of quirks. Both do not have a good history of migration support. (Partly due to the CORBA standard evolving very rapidly).
2. In house expertise.
3. The cost of the solution may be too high. (Need to investigate on this).
4. May a very complicated solution for a simple problem.

Exhibit C18

eStream 1.0 High Level Design

Version 0.3

[redacted]

Notes

The following is roughly what's changed since the last version (0.2):

- The functional requirements and use cases have been removed. These will be documented in the eStream Requirements Document in future revs.
- The entire accounting hierarchy (what is a user and account, how are they grouped, at what level does billing take place) is undergoing revision, and has been removed from here for this version.
- Component descriptions should be more consistent now.
- The database of user and subscription information in the client block diagram has been removed. See the notes below.

Known issues

- The mechanism for how a pathname on a client machine translates into a globally unique FileID for any eStream server is unclear. This is a major design issue that crosses many components on both the client and the servers.
- The accounting hierarchy and its impact on this design are missing.
- If and how copy-on-write will work for writes to the Z file system is quite unknown.
- Which server manages user/account/group/subscription data is quite uncertain. Representing this by a data cylinder was wrong, and I removed this. However, all the interfaces specified below for an "ASP web server" are now just plain wrong, and the server team needs to suggest the appropriate changes to the HLD for the server topology.
- The "Server Data Objects" section at the end of this document needs to be rewritten, in terms of interfaces that client and server components supply to support these data.

Introduction

This document describes the high level design for the eStream 1.0 product. The organization used is:

- Definitions
- Block diagrams for both the client and server portions, showing all major components
- Each component, generally broken down by
    - purpose

- functionality
- global data managed, if any
- interfaces for use by other components

To understand the problem being solved in this design, see the "eStream Requirements Document" for information.

Definitions

account

A billing entity consisting of a set of users and subscriptions

user

An entity authorized to use an account

subscription

An agreement between user and the ASP to use an application under terms of licensing.

license

Legal right to use an application at any given time.

account admin

A special kind of user who can add/delete other users from an account.

server admin

Administrator for all the eStream servers and database.

AppID

A unique representation of an application. There is one to one mapping of AppIDs to apps.

FileID

Within an application, a unique representation of a particular file.

access token

This represents the right to run an eStreamed application. The client must acquire an access token before accessing any file (e.g., executing) in an eStreamed app.

application

An *application* is the set of all files and directories, served by an eStream server, that make up a subscribed application. For example, any executable file, DLL, icon file, or data file associated with an eStreamed version of FrameMaker is part of this application.

application installation

This is the process of locally installing all bits necessary to execute an application via eStream. Most files in the application can be read or executed via the eStream file system; some must be installed locally. Some configuration data must also be downloaded and processed to allow seamless execution of apps.

app install block

This is what needs to be downloaded and installed during application installation. It might consist of:

- all configuration files that must be installed on the client machine
- all registry spoofing information required to run the app
- all file spoofing information required to run the app
- the names of all files and directories that make up the application
- initial prefetch data
- initial pages for critical application files

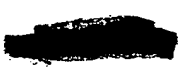
It's quite possible that this app install block is actually an executable file or a DLL that performs all actions to make an application ready to run, rather than simply a block of data.

ASP ID block

An *ASP ID block* consists of all the information about the applications available to a given user for a given ASP, on a given client machine. Since a user might belong to multiple accounts for an ASP, this represents all subscribed applications for all accounts for that user.

Such data might consist of:

- user name
- password
- ASP contact info (IP address, URL, etc.)
- list of subscribed apps
    - is the app installed on this machine?
    - serial number for app
    - ADRM server(s) to use for validation of this app
    - App server(s) to use to retrieve the app install block



  - App server(s) to use to retrieve app file data
- last time stamp when the ASP was checked for new subscriptions

client certificate

The *client certificate* for a client machine is a digital signature used to identify it to eStream servers. We anticipate this to be used for requests that don't require an access token -- i.e., a valid license. For example, retrieving the app install block, or data for eStream application files that don't require license validation.

client machine

This is a computer on which an eStreamed application executes. It may host multiple registered users, and a single user can install the eStream client on multiple client machines.

eStream client

This is the aggregate of all the software required on a client machine to subscribe to, install, and execute an eStreamed application.

eStream file system

The *eStream file system* (or EFS) is a distributed file system with prefetch and caching functionality. All file data and metadata accessed through the EFS is subject to license validation before being available from a server.

license validation

The act of validating a license means gaining an access token. Generally, before an eStream application can be run on a client machine, the validity of using this application by the current user must be checked. This check is done when certain files associated with the application are accessed; an eStream server is contacted to perform this check and return an access token.

subscription serial number

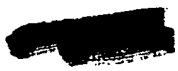
Each application that a user is associated with a serial number. This identifies *both* the application and the user uniquely, and hence can be checked easily during license validation.

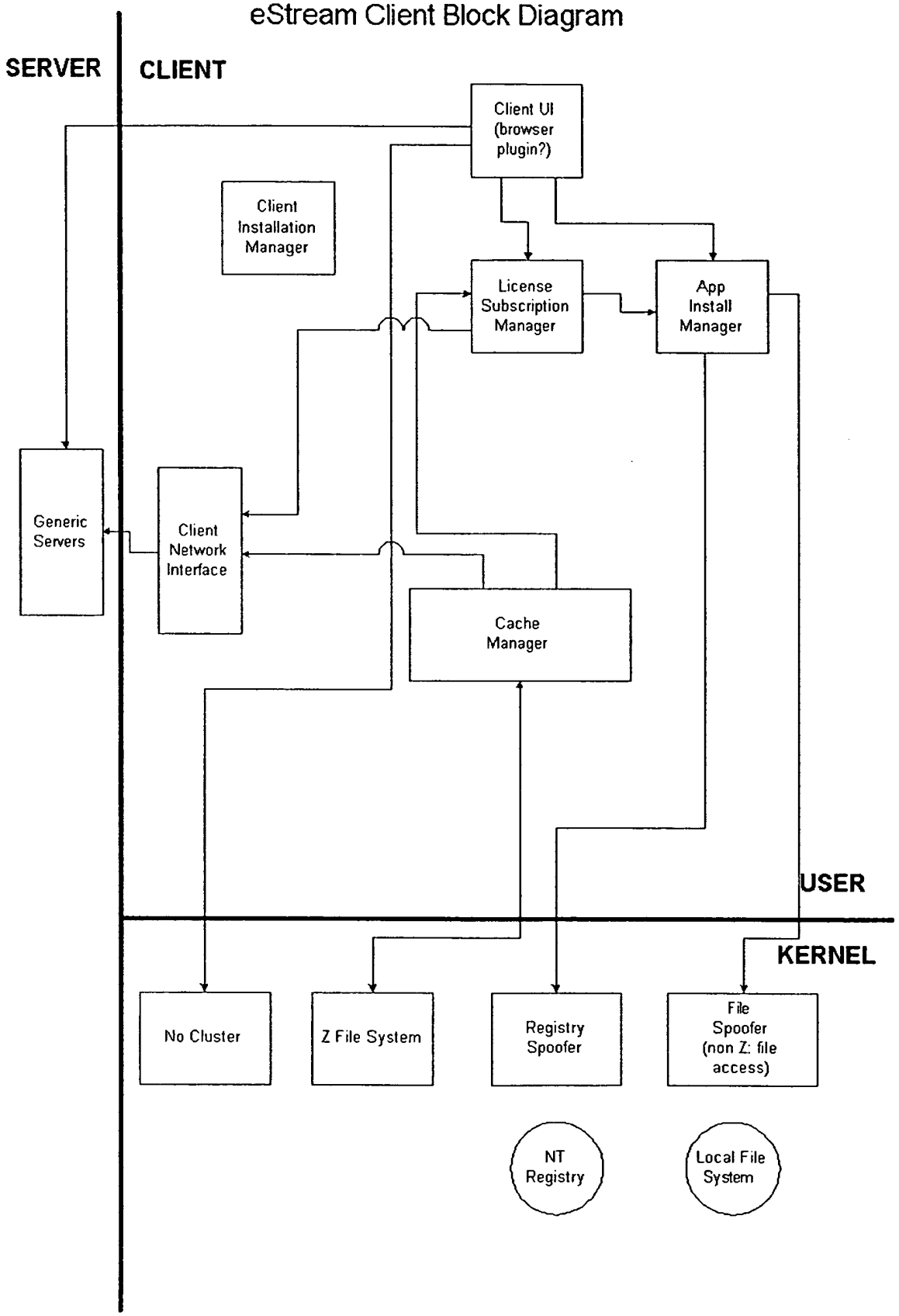
Block diagram

The following are simple block diagrams of the client and server components. Some conventions:

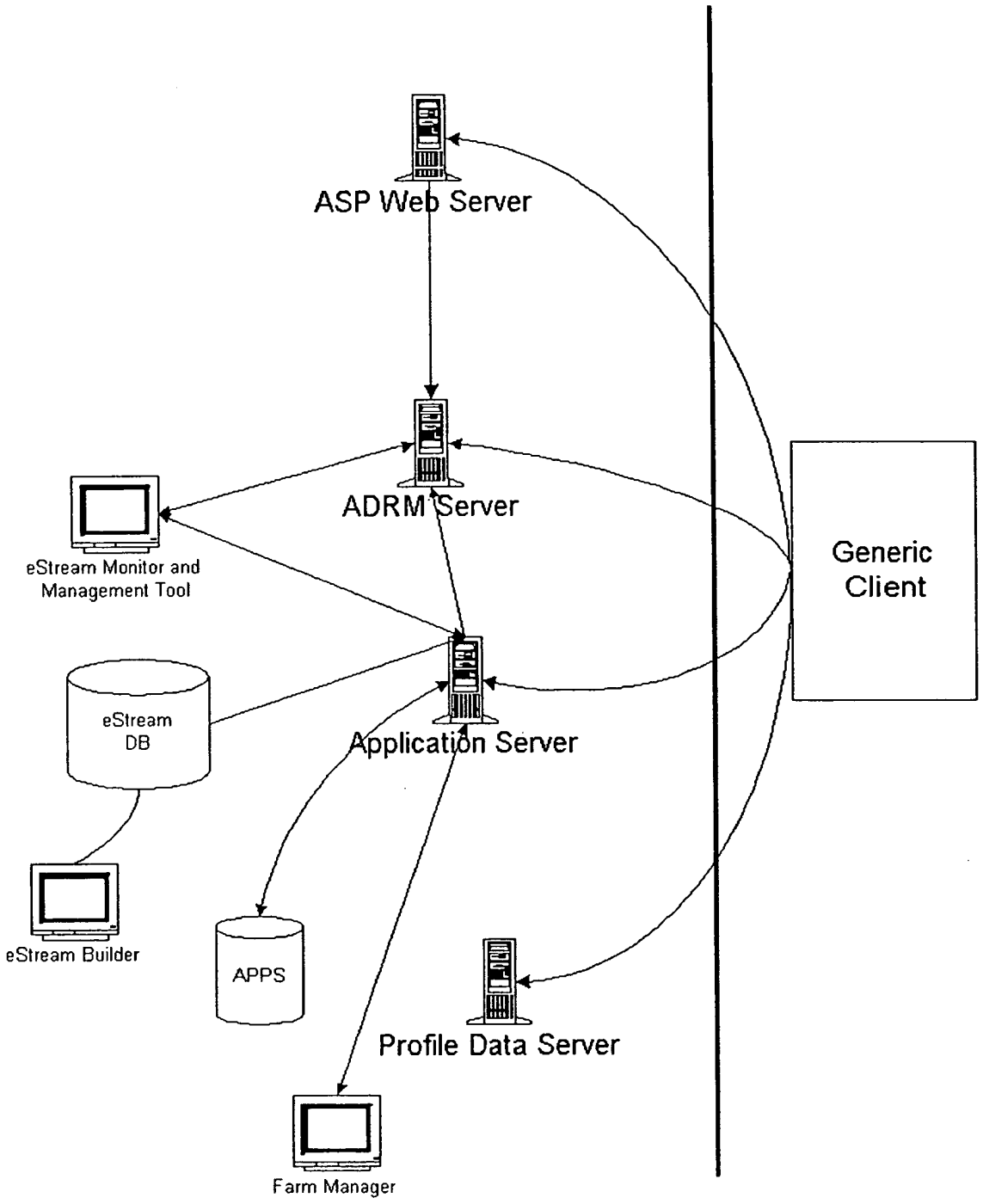
- A box represents a **logical eStream component**. A component may exist as a distinct process, or it may be grouped with other components into a common process.
- A line between components represents an interface call from one to another. If A calls B, there's a arrow on the end of the line at B. If A and B call each other, there's an arrow on both ends of the line.

Note that data stores are **not** represented in these diagrams; if a data store is centrally managed, then there is a component that has interfaces to allow access to these data.




eStream Client Block Diagram
SERVER
CLIENT
Client UI (browser plugin?)
Client Installation Manager
License Subscription Manager
App Install Manager
Generic Servers
Client Network Interface
Cache Manager
USER
KERNEL
No Cluster
Z File System
Registry Spoofer
File Spoofer (non Z: file access)
NT Registry
Local File System

eStream Server Block Diagram



Component descriptions

Client components

The client components are all identified in the block diagram above. Very briefly, some points:

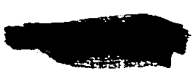
1. A web browser on the client machine will be used for most user interface requests: subscribing to applications, requesting subscription and payment information, and so forth. Configuration of the eStream client software will be done using a UI which may be different from a web browser. Some thoughts on this are listed below.
2. The eStream cache manager is the heart of the client software, and is the component that actually requests file data from the servers.
3. The license subscription manager has the task of tracking all valid subscriptions to applications from an ASP, and tracking which applications have, or need, a license validation to access files.
4. The app install manager's task is to wait until it's told to install a newly subscribed application, and then do so. It also keeps track of what needs to occur when uninstalling an application.
5. The client network interface simply takes requests from the rest of the client components, and forwards them on to the appropriate eStream servers.
6. The eStream file system (EFS, aka the "Z" file system) is a standard kernel-mode network redirector. It presents the normal FS interface to the rest of the NT executive, and requests data from the eStream cache manager to satisfy requests made of it.
7. The registry and file spoofers are kernel-mode drivers that monitor registry calls and file open requests, respectively.
8. The No Cluster component is a very simple kernel-mode driver that disables page clustering for reads.

Installation Manager

Purpose

Installation of the eStream client software is not different than installing any other client software package such as Winzip or Office. The eStream client installation is separate from the installation and configuration of eStream subscribed applications. Some of the possible pieces that eStream would need to be installed are listed below.

1. Device Drivers
2. Applications Executables
3. Application Components
4. Shared Components
5. Registry entries



6. Shortcuts and Start menus
7. Help Files
8. Uninstaller

Once the pieces of the application to be installed are brought together then an install program must be constructed.

Functionality

The Installation Manager consists of the following sub-components

Installshield

Installshield is the industry standard for building installation sets for Microsoft Windows. Installshield will take a set of executables and data files and create a media installation. The Installshield environment provides a scripting language that will allow a high degree of customization of target installation. The essentials issues for any installation are.

1. How much of the application does the user wish to install?
2. Is the users system capable of running the application?
3. Where does the user wish to install the application?
4. Does the user have enough space to install the applications?

Installshield has a wizard that will set up a project. When the install shield program is compiled a media must be specified. The most common media types are floppy, CD Rom, and Web media builds. For eStream we may have to ask the clients to reboot the machine since we are installing kernel mode components that might need a reboot to take effect.

Install From the Web

This program is another product that is sold by Installshield that will take a complete installation set and create a single executable .exe that can be easily downloaded from a web site.

Uninstaller

Installshield will provide an uninstaller when it builds the install program.

Registry Settings

There are three ways that Installshield can patch the system registry.

1. Run regsvr32.exe on self-registering .dll files. When the uninstaller is run it will use regsvr32.exe /u to un-register the .dll file.
2. Patch the registry statically.

3. Patch the registry based on Installation Options from the install shield script program.

Artwork

The Installshield program for eStream will require a splash screen and possibly one or two other artwork components.

eStream Client UI Module

The eStream Client UI module is a client component, currently expected to be running in user space.

Functionality

The eStream Client UI module supports reporting eStream-specific error & informational messages to the client user and soliciting replies when appropriate. It allows the eStream client user to view and change the list of applications currently installed on the client system and the list of ASP accounts currently known to the client system.

Interfaces

ReportMessageToEStreamClientUser(IN message)

Display specified message in EStream Client UI message window.

QueryEStreamClientUser(IN message, OUT response)

Display specified message in EStream Client UI message window and solicit yes/no response for return to caller.

Installed Applications UI

The Client UI interface allows the user to request that the list of the applications currently installed on the client be displayed. The Client UI Module gets this list by calling AIM/GetAppInstallList().

The Client UI interface allows the user to select an application from this list to be uninstalled. The Client UI Module calls AIM/UninstallApp() to accomplish this.

The Client UI interface allows the user to enter the information necessary to get a new application installed. The Client UI Module calls AIM/InstallApp() to accomplish this.

The Client UI interface allows the user to request that the list of applications currently installed on this client be exported to a file, in a form which would allow that list to be

imported on another client. The Client UI Module calls AIM/ExportInstalledApps() to accomplish this.

The Client UI interface allows the user to request that a list of applications that was exported by eStream running on another client be imported from a file & installed. The Client UI Module calls AIM/ImportAndInstallApps() to accomplish this.

Known ASPs UI

The Client UI interface allows the user to request that the list of ASPs currently known to the client be displayed. The Client UI Module gets this list by calling ???.

The Client UI interface allows the user to select and connect to an ASP in the list. The Client UI Module accomplishes this by ???.

The Client UI interface allows the user to select an ASP from this list to be deleted. The Client UI Module calls ??? to accomplish this.

The Client UI interface allows the user to enter the information necessary to record information about a new ASP account. The Client UI Module calls ??? to accomplish this.

The Client UI interface allows the user to request that the list of ASPs currently known to this client be exported to a file, in a form which would allow that list to be imported on another client. The Client UI Module calls ??? to accomplish this.

The Client UI interface allows the user to request that a list of ASPs that was exported by eStream running on another client be imported from a file & installed. The Client UI Module calls ??? to accomplish this.

eStream Cache Manager

Purpose

The eStream Cache Manager (ECM) is a client component, currently expected to be running in user space. Its goal is to:

- Handle all file requests from the eStream file system, either by using previously cached contents or requesting the contents from a server.
- Intelligently use prefetching of file data to reduce latency of pages requested from the EFS.
- Work with the license subscription manager to insure that all applications have appropriately validated licenses before their files are accessed.

Functionality

The ECM handles the volatile & non-volatile eStream cache on the client machine. It performs demand fetching and prefetching from the appropriate server(s), using profiling data or heuristics. Based on the client's observed behavior, it compiles updated profiling data, which may periodically be uploaded to a server.

Interfaces

The ECM takes requests from the EFS driver, and makes requests to the client network and LSM modules.

In the descriptions below practically every call could fail for a variety of reasons. The associated error handling paths are not shown at this level of the design.

Open/Create(IN Filename, IN FileOptions, OUT Handle)

Called from the EFS.

This does the following basic tasks:

- If the filename and mode options correspond to a FileID that is already known and legal to use (from it's cache), it can just return the handle for this file.
- Otherwise, it must ask the LSM for an access token for this file. (This request may simply return an access token previously created for other files making up the application.)
- Launch any prefetching and/or active cache loading activities desired for the new app.
- Request a file handle from an appropriate app server, via the client network component.
- Return the handle to the caller

Close(IN Handle)

Called from the EFS.

This basically:

- Informs the LSM that the file is being closed
- Unloads (or marks as victims) app's cached entries as desired

Read(IN Handle, IN ReadOffset, IN ReadLength, IN BufferPtr, OUT BytesRead)

Called from the EFS.

This does the following:

- Update the profile data for this file
- Check the cache for the requested data
- If not there, request the appropriate pages from an app server, along with any page prefetches that are needed, and place the retrieved data in the cache
- Fill in the buffer and return the number of bytes written to this buffer

Write(IN Handle, IN Buffer, IN Offset, IN Length, OUT BytesWritten)

Called from the EFS.

This will use some copy-on-write scheme. It may be as simple as locking the written structures in the cache.

Global Data

ActiveAppsData Structure

Table of data with an entry for each application that is currently active on the client. Fields for each entry are listed below.

- AppPrefix
- AccessToken
- ServerName
- FilesOpen
- LocalPathName

FileID Type

A globally unique identifier defined for each file associated with an eStream application. All EStream-managed files have these identifiers to allow a common & unambiguous method of file referencing between clients & servers & to simplify switching the client to an alternative server.

ProfileData Structure

Exact contents of this data structure will be defined in the low level design phase; at this point, assume predecessor/successor pairs w/counts.

Volatile & Non-volatile Caching Structures

Exact contents of these data structures will be defined in the low level design phase.

License Subscription Manager (LSM)

Purpose

The LSM tracks current subscription information and determines the need for license validation. It is informed of subscription changes from the client UI, and is queried by the ECM to validate accessibility to different applications, based on the license model for the subscription to that application.

Functionality

The LSM tracks the users subscriptions to different ASPs; it is part of the client component downloaded on a client machine. The LSM starts running when the client component starts running, and is remains active until it stops.

The LSM has a few major tasks:

1. Keep track of what subscriptions the current user has available from all ASPs
2. Determine which application a given file is a part of
3. Acquire an access token to validate a license for file requests that require one

There are two ways that the LSM updates its list of known subscribed applications:

1. It may be informed of new subscriptions, or of applications that are unsubscribed, by the client UI, as part of a browser plugin in conjunction with an ASPs web site.
2. It may asynchronously poll an ASPs ADRM servers to get updated lists of subscribed apps.

When the users start running any of the subscribed eStream applications—i.e., when any eStream'ed file is opened—the ECM queries the LSM before servicing any requests. The LSM checks to see which subscribed application this file belongs to, and, if necessary, gets the appropriate access tokens from ADRM servers along with the identities of application servers that can be used to run the applications; it uses the client certificate obtained when the connection to the ASP was made. At the same time, the LSM can decide to cache the access tokens and the identities of the application servers and decide to serve them directly from its cache.

The ECM informs the LSM when files open and close, and determines from this when applications start and end. The LSM keeps track of when access tokens are expiring and can request for additional access tokens when applications are running and the current one is about to expire.

Global Data

The global data managed by the LSM includes

1. The ASP ID Blocks which are obtained when the user on the machine establishes a connection with an ASP from which the user has subscribed applications.
2. The access tokens and the identities of the applications servers that are obtained from the ADRM servers when the user tries to run the applications.

Interfaces

The LSM exposes the following set of APIs to the client UI:

SubscribeApp(IN ASPId, IN AppID, IN LicenseInfo)

This routine in turn will call the App Install Mgr to install the application on the client machine. This will return a Boolean stating success or failure.

UnsubscribeApp(IN ASPId, IN AppID)

This routine will NOT implicitly uninstall the application. Applications must be explicitly uninstalled. This will return a Boolean stating success or failure.

GetAppList(OUT SubscribedAppList)

This routine will return a pointer to a list of subscribed applications on the client machine.

The LSM exposes the following set of APIs to the ECM:

CheckAccess(IN Path, OUT Root)

The LSM establishes a correlation between the Path and the AppID by querying the App Install Mgr. This routine in turn may contact the ADRM server for appropriate access tokens. This will return a Boolean stating success or failure. At the same time Root will get set to the head of the path that identifies the application so that the file system can use the same access token for everything under "Root".

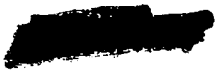
BeginApp(IN AppID)

To indicate the start of an application. **Note:** this may happen implicitly during CheckAccess().

EndApp(IN AppID)

To indicate the end of the application. **Note:** this may happen implicitly during CheckAccess().

The LSM makes the following API calls.



1. InstallApp(ASPId, AppID) to the App Install Mgr to install the subscribed applications.
2. GetAppId(Path, &Root) to the App Install Mgr to get the AppId from the Path. "Root" is explained above.

The LSM sends messages to the ADRM server for getting access tokens. When a user goes to a new machine and installs the eStream client, the LSM obtains the subscription information from this server when the user first establishes a connection with it.

Application Install Manager (AIM)

Purpose

The AIM is the contact point for installation and uninstallation of applications on a client machine. It gets the requests from the LSM to install applications when the user subscribes to applications, and it gets requests from the Client UI to uninstall applications.

Functionality

The AIM manages application installs on the client machine. It keeps track of what applications have been installed on the client machines, where they have been installed and the various components that are part of the installation. It contacts the application servers (obtained from the ASP ID block) to get the AppInstallBlock. This may be a data block, an application or a dll. The AIM uses the AppInstallBlock to then make the appropriate calls to the Registry and File spoofers; to install some files on the local disk; to "warm" the cache and to update the start menu and other short cuts as needed.

Global Data

The Global Data managed by the AIM includes –

1. The AppInstallBlock obtained from the app server that is used to do the installation.
2. The AppID->Path co-relation that is required to check for access privileges.

Interfaces

The AIM exposes the following interfaces –

InstallApp(IN ASPId, IN AppID)

To install the application using a specific ASP server to get the AppInstallBlock.

UninstallApp(IN AppID)

To uninstall the application from the client machine.

GetAppId(IN Path, OUT Root)

To return the AppID given the Path that is being used to open a file/directory on the eStream file system.

GetAppInstallList(OUT InstalledAppList)

To get a list of the applications currently installed.

The AIM makes calls to the registry and the file spoofers using the AddRegSpoofEntry, AddFileSpoofEntry, etc. APIs.

eStream client network component

This section deals with the components that communicate with the servers.

Purpose

The client network component is the common point of connection between the rest of the eStream client components and the various eStream servers. Any client module that calls an interface of a server does so through the network component.

This component is basically stupid. It knows the protocols needed for communicating with the various servers, and it can encode the requested messages via these protocols, but it doesn't try to be smart with regard to failover, or authentication rejection, or other error conditions. The network component lets its caller deal with such matters.

One design assumption here is that data is received from an eStream server only in response to a request it has made of this server. In other words, all requests originate with the client, never from the server.

Functionality

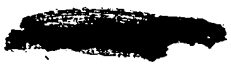
The client network component communicates with the following servers for the types of requests listed.

ADRM server

1. Validate a user for this ASP and get subscription information
2. Validate a license for a subscribed app

App server

1. Open a file/directory for a subscribed app



2. Various file requests on a previously opened file/directory

Global Data

Probably none (?).

Interfaces

ValidateUser(IN ADRMServer, IN AspAndUserData, OUT SubscriptionInfo)

This interface is called by the LSM; it is used both to validate a user and get updated subscription information for a given ASP.

ValidateLicense(IN ADRMServer, IN APPId, IN ClientCertificate, OUT AccessToken, OUT AppServerList)

This is called by the LSM, to get an access token for an application before its file can be accessed.

AppOpenFile(IN AppServer, IN AccessToken, IN FileDesignator, OUT Handle)

This is called by the ECM, to for any eStream file. Note that this is also used to retrieve an AppInstallBlock, when requested via the AIM. **Note:** the FileDesignator is still undergoing design.

AppReadFile(IN AppServer, IN AccessToken, IN Handle, IN OUT Buffer, IN Offset, IN Length, OUT BytesRead)

This is called by the ECM.

UploadAppProfileDataRequest(IN ADRMServer, IN ProfileData, OUT Success)

It's unclear who calls this!

eStream File System Driver

The file system driver interfaces with the operating system's installable file system facilities, forwards file system requests that it cannot directly satisfy to the ECM, and uses the NT File Cache to optimize repeated accesses to the same data. This component will be very operating system specific, while the interfaces it exposes to the cache manager will be (mostly) OS independent. The file system driver resides in kernel space and implements a portion of the entire eStream file system. Other components, such as the cache manager, the client network interface, and the app servers, implement the rest of the eStream file system. These other components are not necessarily kernel-mode resident.

Functionality

The eStream file system driver (EFS) will send most requests from the operating system to the cache manager. It will interface with the standard NT File Cache Manager to avoid sending redundant requests to the cache manager. And it must support functionality for the ECM to notify it when the data structures it has cached have become invalid.

Global Data

The only globally-visible data managed by the file system driver are various things that it may cache. This includes both file data pages as well as directory contents. These data are relevant to the ECM, because it may want to invalidate the contents of the caches if it finds a newer version of a data page or finds that (visible) directory contents have changed.

Interfaces

These are the logical interfaces that are exposed to the ECM. The EFS has standard file system interfaces that are used by the NT Executive, but these are not listed here.

InvalidatePage(IN FileHandle, IN PageOffset)

Invalidates the specified page for the specified file handle in the cache.

InvalidateDirectory(IN DirHandle)

Invalidates the specified directory's contents in the cache. This may result in the eStream file system driver sending directory change notifications.

ShutdownFileSystem(IN Force)

Attempts to shut down the file system. If Force is true, the file system will be shut down regardless of whether any processes still have handles that are open on the file system. If Force is false, this routine will return an error if there are any open file handles. After the file system is shut down, any attempt to access the file system will result in errors rather than being forwarded on to the cache manager, until StartFileSystem is called.

StartFileSystem()

Causes the eStream file system to begin accepting requests and forwarding them to the eStream cache manager.

Virtual Memory Clustering Disabling Driver

Purpose

The VM clustering disabling driver (aka NoCluster) disables virtual memory clustering under Windows. While we don't fully understand all the implications, using this driver substantially reduces the average file system paging request size and can dramatically improve performance of eStream, especially on slower connections.

Virtual memory clustering, as implemented in Windows NT/2000, is intended to improve performance when paging to and from physical disks. If possible, we would like to disable clustering only for those threads/processes that will be doing a significant amount of I/O to the eStream file system.

Functionality

The VM clustering disabling driver maintains a set of criteria for threads whose clustering should be disabled. It will make decisions about which threads should have clustering disabled without contacting any other components.

Global Data

The only data managed by this component are the criteria for selecting threads whose clustering should be disabled. In the simplest implementation, this would be nothing, and the driver would disable clustering for all threads. Whatever tables it uses will be resident in the kernel, and the driver will be able to access them as needed without making calls to a user-mode component to read them.

Interfaces

The interfaces for this component are minimal. Clustering may be enabled or disabled, and the criteria for which threads to manipulate can be changed.

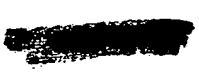
StartDisablingClustering()

This interface notifies the driver that it should begin disable virtual memory clustering, using the currently specified criteria.

StopDisablingClustering()

This interface notifies the driver that it should stop disabling clustering. Note that due to implementation problems, we do not support actual unloading of the driver, though it can be removed from the system by a reboot.

ChangeClusteringCriteria(IN Criteria)



This interface allows the caller to cause the driver to change the criteria it uses to select threads it should manipulate. The criteria might be specified in the interface, or it might specify a file that the driver should read for the new criteria.

QueryClusteringCriteria(OUT Criteria, OUT Active)

This interface allows the caller to find out what criteria the VM clustering disabling driver is currently using, and whether or not it is currently active.

File Spoofer

Purpose

The purpose of the file spoofer is to redirect file system accesses from some non-eStream drive. This may be necessary in order to support applications running under eStream that are hard-wired to access files in a specific location. The file spoofer may also be used if we are interested in providing a version of some system file different from the one actually on the client machine.

Functionality

The file spoofer will intercept File Create calls for files that we are interested in spoofing and ensure that these creates are redirected to a file we specify. The redirection could be to a file on the Z file system, or to another, non-eStream'ed file.

File open is a very common occurrence, so the file spoofer must operate quickly. The file spoofer should maintain in-kernel whatever data structures it needs to make a spoofing decision.

Global Data

The file spoofer must maintain a database indicating which files to spoor, which file to replace them with, and possibly which processes should be spoofed. All of this information must be kept in-kernel so spoofing decisions can be made quickly. This database may change depending on which eStream apps are currently installed or running.

Interfaces

StartFileSpoofing()

Causes the file spoofer to begin file spoofing, using the current spoof database.

StopFileSpoofing()

Causes the file spoofer to stop spoofing, but does not change the spoof database.

AddFileSpoofEntry(IN SpoofEntry)

Adds an entry to the spoof database. It is not necessary to stop and restart the spoof database to add an entry.

RemoveFileSpoofEntry(IN SpoofEntry)

Removes an entry from the spoof database. It is not necessary to stop and restart the spoof database to remove an entry.

QueryFileSpoofDatabase(OUT SpoofEntryList)

Queries the current contents of the spoof database.

ReplaceFileSpoofDatabase(IN SpoofDB)

Replaces the entire spoof database. It is not necessary to stop and restart the spoof database to perform this action, and it is considered atomic.

1.4 Registry Spoofer

Purpose

The purpose of the registry spoofer is to provide to eStreamed and other applications registry entries for eStreamed apps, and to capture registry writes by eStreamed apps so they can be purged from the registry or shipped to other clients for configuration ubiquity.

Functionality

The registry spoofer must redirect registry reads and writes. Because registry accesses are quite common, the redirector should be able to service registry spoofs without forwarding the requests to a user-level process.

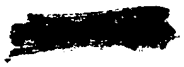
Global Data

The registry spoofer maintains a spoof database of which registry entries to spoof, and which processes to spoof them for. This database should be kept in-kernel so that the spoof decisions can be made quickly. The spoof database may change depending on what eStream applications are currently installed or running.

Interfaces

StartRegSpoofing()

Causes the registry spoofer to begin spoofing using the current database.



StopRegSpoofing()

Causes the registry spoofer to stop spoofing. This does not change the registry spoof database.

AddRegSpoofEntry(IN SpoofEntry)

Adds an entry to the registry spoofing database. It is not necessary to stop spoofing in order to do this.

RemoveRegSpoofEntry(IN SpoofEntry)

Removes an entry from the registry spoofing database. It is not necessary to stop spoofing in order to do this.

ReplaceRegSpoofDatabase(IN SpoofDB)

Replaces the entire registry spoofing database. It is not necessary to stop spoofing in order to do this, and it is considered an atomic operation.

QueryRegSpoofDatabase(OUT SpoofEntryList)

Queries the contents of the registry spoof database.

Server components

The servers described below are **logical servers**. Note that a single server machine can serve all functions for a small ASP; alternatively, farms of servers can be used to provide the functionality of a single logical server.

NOTE: The distribution of servers and the functionality provided by them are somewhat uncertain to date. In particular, exactly who manages the actual accounting, user, and group data is undecided. The group producing the LLD for the eStream servers need to flesh this out. For now, this document assumes that the ADRM server ultimately manages these data, and supplies interfaces to callers to access these data.

The servers described are:

1. An ADRM server handles user/account/subscription data management, as well as validating licenses.
2. An ASP web server is a front-end for requests to add users, subscribe to applications, and do various user and application level queries. Generally this forwards these requests to an ADRM server.
3. An application server handles requests to open and read eStream files.
4. A profile data server will receive uploaded profile data from a client machine to enable better initial profile and prefetch maps in eStream sets.

ADRM Server

Purpose

The Account/Digital Rights Management (ADRM) server is responsible for:

- Managing data related to users, the groups they belong to, and the applications they are subscribed to
- Validating the licenses for applications executing on clients
- Tracking all outstanding licenses currently in use

Functionality

Client machines sends requests to the ADRM server to add or delete subscriptions, to receive an access token to execute an application, and to manage their account/group/user relationship.

Access tokens have an expiration time, so the client must reacquire them at regular intervals. When an eStreamed application exits, the client informs the ADRM server to release the access token. Any outstanding access token not released or reacquired within the expiration time will be automatically released by the server.

Interfaces

AcquireAccessToken(IN UserInformation, IN AppId, OUT AccessToken, OUT AppServerList)

This is called by the eStream client to gain validate a license before executing an application.

This is used to insure that a user has the right to use a particular app in a subscription from a specific account. The server returns an access token and a list of app servers from which the client can access the application file data. If the user doesn't have a valid license to use the requested application, a failure message is sent to the client. The server writes the start time of this application usage into the database for billing processing.

RenewAccessToken(IN OldAccessToken, OUT NewAccessToken, OUT AppServerList))

Note: This may just be replaced by AcquireAccessToken()

This is called by the eStream client.

The server receives a message from a client to renew its access token before the expiration of the token. The server returns a new access token and a list of app servers. This allows the server to redirect the client to a different app server in case it knows of changes to the list of available servers. Once the token is expired, the ADRM server

writes the end time of this usage information into the database and the client must reacquire the access token before files for this application are available to it.

ReleaseAccessToken(IN AccessToken)

This is called by the eStream client.

The client returns the token to the server when the eStream app terminates so other clients can acquire the token. The server writes the end time of this usage information into the database for billing processing.

AddApplicationServer(IN AppServer, IN ApplicationList)

This is called by an application server.

The ADRM server is informed of the availability of a new application server. The ADRM server adds this new app server to its list of app servers.

RemoveApplicationServer(IN AppServer)

This is called by an application server.

The ADRM server is told of the removal of an app server. It must remove this app server from its list of such servers to prevent any clients from using that server.

AddApplicationServerApplications(IN AppServer, IN ApplicationList)

This is called by an application server.

The ADRM server is informed of the availability of a new application on a given app server. The ADRM server adds this new app to the list of applications that the server has available.

RemoveApplicationServerApplications(IN AppServer, IN ApplicationList)

This is called by an application server.

The ADRM server is told of the removal of an application for an application server.

GetTrafficHistory(IN TimeCriteria, OUT HistoryData)

This is called by a server UI tool, or possibly by other ADRM servers.

The administrator monitoring, reporting, and management tool UI program can query the ADRM server for load information. The server logs all client requests to acquire access token. This raw information can either be sent directly to the UI program or it can be preprocessed before sending to the UI program.

GetErrors(IN TimeCriteria, IN ErrorType, OUT ErrorList)

This is called by a server UI.

The admin UI program can query the ADRM server for any errors. The errors can be categorized by type of errors or errors that occur between certain time periods. A small sample of the possible ADRM server errors includes: client access token timeout, failure to read user information from the account database, failure to get the license information, failure to write usage information into the database, etc...

GetIllegalAccesses(IN TimeCriteria, IN AccessType, OUT AccessList)

This is called by a server UI.

The admin UI program can query the ADRM server for any illegal accesses. The illegal accesses can be categorized by type and time period. A small sample of the possible ADRM server errors includes: failure attempts to access ADRM server with bad password repeatedly in a small time period, failure attempts to use a particular license, any access attempts from a non-typical IP address ranges for a particular account, etc...

GetAppID(IN AppName, OUT AppID)

This is called by a server UI.

Returns a unique identifier associated a particular application

SetApp(ID IN AppName, IN AppID)

This is called by a server UI.

Stores a unique identifier associated a particular application

Application Server

Purpose

The application server is there to handle read requests for files accessed by eStream clients. Any file accessed on a client through the EFS can have this read request passed to an app server.

Functionality

This will be the hardest working eStream server. It will respond to both synchronous (demand fetching) and asynchronous (prefetching) page requests from many different clients, for many different types of applications and files within those applications.

Interfaces

GetFileInfo(IN AccessToken, IN FileID, OUT FileInfo)

This is called from an eStream client. Given any file within an eStream application, return metadata about it. The access token is provided for validation.

ReadFile(IN AccessToken, IN FileID, IN Length, IN Offset, OUT Buffer, OUT BytesRead)

This interface is called by an eStream client, and will allow the client to access any eStreamed application file and AppInstallBlocks. How the FileID for an AppInstallBlock is achieved is unclear at present.

OpenFile() / GetFileID()

Note: This is a placeholder for an API that may be needed. This depends a lot on the eventual communications between client and server for associating a file pathname with a FileID.

ASP web server

Purpose

This describes, of course, only those interfaces on an ASP web server that relate to handling eStreamed applications.

Logically, the ASP web server is the backend web interface for user requests—e.g., get billing information, subscribe to a new app, or request a list of all possible apps a user can subscribe to. In the current model, the web server doesn't actually handle these requests, but instead passes them on to the appropriate eStream-centric server.

NOTE: The following interfaces are not updated from the previous version! They were written with the assumption that the web server actually manages all the data described above. We need the server team to suggest the changes that should take place here!

Functionality

Interfaces

AddADRMServer()

The ASP Web Server is informed of the availability of a new ADRM server. The ASP Web Server adds this new ADRM Server to its list of online ADRM Servers. Periodically, the ASP Web Server can query the list of ADRM Servers for its load

information. When a new client connects to the ASP Web Server, the client can be informed of the subset of ADRM Servers with the least load.

Callers: ASP Web Server

Input:

- ADRM Server IP
- list of Account Ids that the ADRM Server supports.

Output:

- success or failure

RemoveADRMServer()

The ASP Web Server is told of the removal of an ADRM Server. It must remove the ADRM Server from it's locally cached list of ADRM Servers to prevent any future clients from using that particular ADRM Server.

Caller(s): ASP Web Server

Input:

- ADRM Server

Output:

- success or failure

ValidateSubscribedUser()

Inputs:

- SubscriptionToken

Outputs:

- success/failure

Validate subscribed user is called by the ADRM server to check out a license.

- Find account(accountNumber)
- Find user(username)
- Find license(SubscriptionID)

If a license is found to be available, check it out and return OK else return NO_LICENSE_AVAILABLE.

Add/RemoveAccount()

Input:

- ownerUsername
- ownerPasswd
- billingInfo

Output:

- accountNumber

Creator must supply info for the first user, who is also the account owner.

Add/RemoveSubscribableApp()

Input:

- application
- AppServer locations
- name description

AddAccountUser()

Input:

- account number
- user name
- initial password

Add/Remove/Increment/DecrementFloatingLicense()

Input:

- account number
- list of users
- subscription
- number available

Add/RemovePerUserLicense()

Input:

- account number
- user
- subscription

CompileAccountUsage()

Input:

- account number

Add up and report all the usage by members of the account.

ClearAccountUsage()

FreeLicense()

Input:

- user
- subscription

Frees a license that had been previously checked-out by a user.

GetUserPriveleges()

Input:

- account number
- username

Check privileges of a logged in user for the purposes of allowing user/subscription management and other account changes

ShowCheckedoutLicenses()

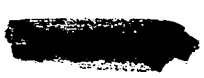
GetAccountInfo()

Input:

- account number
- username

ListPossibleSubscriptions()

Input: none



ListCurrentSubscriptions()

Input:

- accountNumber

CreateSubscription()

Input:

- account number
- license data (depends on license type)
- ApplicationPackage

Output:

- subscriptionID
- ADRM server names.

Builder

Does not talk to any other module. Probably an associated set of tools managed via a script plus manual procedures that create intermediate data files, and finally produce the eStream set.

Farm Manager

Simply takes user commands and input, activating the following functions on the actual Application Server process:

StopServer(boolean graceful)

ConfigureServer(config_parameters)

EnableEstreamSet(appID) - informs the app server that the set is ready to be served

DisableEstreamSet(appID) - opposite of above

Functionality required for upgraded Estream sets may extend this, management of farm-level stuff would be an extension of the above. Synchronizing the availability of apps between the Application Server and ADRM/Account DB must also be handled; at least initially, a human administrator should be able to flips the final switch.

Monitor

The monitor utility is responsible for monitoring the overall health of the system. It is responsible to report server status, server traffic, illegal access etc. It will ping the Application Server and the ADRM server to gather the statistics and display them.

Server data objects

This section needs work! What should be here?

eStream Set

What is an eStream set? It consists of:

- Page prediction map, indicates likelihood a page will be referenced successively after another page. Used to enable accurate prefetching by the client.
- Spoofer info, stuff to initialize register & file spoofer to enable application to run on the client. (ITARD & "Spoofed file mapping list")
- Application content. This includes all files of the application in possibly multiple subtrees (stuff from C:, Z:, Common Files, etc.). These are all then placed under a single application root directory which is "mounted" under the eStream file system under the application's directory ("Microsoft Office" or whatever). This directory structure is then processed to create a special eStream metadata file to represent it, and map file names to FileIDs, one assigned to each file for the app, used by the client. All of the application files can then be placed into a single large file with a FileID index in front to allow the **Application Server** to map requested FileIDs to offsets in the large application content image file.
- The above takes care of everything in the AppInstallBlock except for possible COM dlls that might be necessary (TBD what mechanism generates these).

Account/Subscription Database:

Description:

The Account Subscription database manages all the data required to manage accounts, applications and subscriptions for an applications service provider. It is used to verify users and subscription rights, to log usage, and compile billing for application rental. This document describes the data model for the db, the list of accessor APIs, and finally, a list of scenarios that exercise these APIs.

Data Model

The following is a description of the data model for the Account/Subscription Database.

There are two types of records described.

Static Objects are objects that are stored persistently in the Account/Subscription database. The attributes of each record are divided into two categories:

- Owns: specifies data that is actually contained in and managed by the object. These are the attributes that make the object what it is. For example: an account is not an account unless it has billing information. Each item (except the ASP) must have only one owner, but may have many references.
- References: Attributes that create associations to other first-class objects, making navigation from one item to another simpler.

Transient Objects represent data structures that are created in order to pass information from place to another. For the most part, they provide shorthand identifiers for static objects.

Each attribute listed below has the format:

Type: name – (optional) description

For attributes that don't yet have a defined type, the type is *undefined*

Static Data (Database records)

ASP – The top-level container.

Owns:

- List<Account>: accounts - all of the accounts
- List<Subscription>: subscriptions - all of the available subscriptions
- List<Applications>: applications - all of the applications currently served by Application Servers

References:

- None

Account - Collection of attributes that make up a single account:

Owns:

- Number: accountNumber – number that identifies this account
- List<User>: users – all of the account users
- Undefined: billingInfo – Currently undefined type

- List<License>: licenses - all of the licenses (per-user and floating) that are managed by the account
- List<UsageRecord> usage – list of records that describe usage of all account users

References:

- User: owner - a user who created the account – must be one of the users

User – a single user of an Account

Owns:

- String: username
- String: password
- Undefined: Role – specifies permissions on the account, i.e. owner, administrator vs. regular user
- Undefined: UserInfo – Real name, contact info etc

References:

- List<License>: licenses currently held by user
- List<License>: licenses - licenses to which the user has access. These might be either per-user or floating licenses, or a combination of the two. This list might also incorporate a means of specifying a preference – for example if a floating license and a per-user license are available, use the per-user

Application – single application that has been made available on one or more application servers.

Owns

- String: name
- String: description
- Undefined: AppServer location(s) – A location may be a host name that must be resolved by the appserver farm, or may be an IP address.
- Number: AppID
- A list of FileIDs for this app.

References:

- None

Subscription – An application or group of applications that have been made available for rental by a user.

Owns:

- Number: SubscriptionID
- String: name
- String: description

References:

- User: user
- List<Application>: applications - application(s) associated with this subscription

ApplicationPackage – *application(s) that can be rented*

Owns:

- Undefined pricing

References:

- List<Application> applications

License – base for other licenses – All licenses support the same APIs – check out, check in

Owns:

- None

References:

- Subscription: subscription – subscription for which this license grants rights

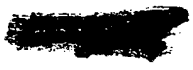
FloatingLicense – one of a fixed number of licenses distributed to a list of users on a first-come first-serve basis. Contains all attributes in License plus:

Owns:

- Number: numTotal
- Number: numInUse

References:

- List<User> holders – the current holders of this license (length = numInUse)
- List<User> allowedUsers – list of users allowed to check out this license



***PerUserLicense** – license tied to a particular user – one desktop (machine) at a time. Contains attributes in license plus:*

Owns:

- Boolean: isInUse
- Undefined – desktopID

References:

- User: allowedUser – the (only) user allowed to pull this license.

UsageRecord – Describes a billable use of the application

Owns:

- Undefined: Start time
- Undefined: End time

References:

- Subscription: subscription
- User: user

Transient Data

AccountNumber – integer that uniquely identifies a single account with an application service provider.

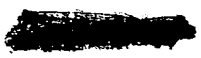
UserName – string that uniquely identifies a single user **within** an account. To uniquely specify a user to an ASP, it is necessary to qualify the UserName with the AccountNumber of which he is a member. All users have

UserVerifier – combination of the AccountNumber, UserName, and UserPassword. Uniquely identifies a user within an ASP

- AccountNumber
- UserName
- UserPassword

SubscriptionID – Integer that uniquely identifies a subscribable application or collection of applications within an ASP.

SubscriptionToken – describes a user-subscription to the ADRM server. Identifies the subscription as well as the user trying to access it.



- UserVerifier
- SubscriptionID

AppID – A unique numeric representation of each eStreamed application. For example, word := 1000, excel := 1001, office := 1002 etc. We should be able to represent software packages using AppIDs.

FileID – Within an eStreamed app, each app-file gets a unique numeric ID.

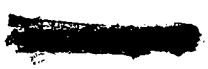


Exhibit D

Estream 1.0 Planning Document

[redacted]

Low-Level Design Status/Plan

Sub Components	Owner	LLD Design Doc completed	LLD review Complted	Estimates for Impl	Impl and Unit Test Completed
Content					
Install Monitor	Sanjay	Done	Done	3 wk	
Builder GUI	Sanjay	Done	Done	1 wk	
FSRFD (Drivers)	Sanjay	Done	Done	2 wk	
AppInstallBlk structure	David	Done	Not needed		
Profiler	David	Done	Done	2 wk	
File Access Monitor	David	Done	Done	1 wk	
Packager	David	Done	Done	1 wk	
eStream distribution	**Bob**	[redacted]	**Status TBD**	**TBD**	
Server Group					
Web Server	Bhaven	Done	Done	8 wk	
Monitor	Mike	Done	Done	4 wk	
SLiM Server	Amit	Done	Done	2 wk	
App Server	Sameer	Done	Done	4 wk	
Admin UI	Bhaven	TBD	TBD	TBD	
End User UI	Bhaven	TBD	TBD	TBD	
Common Server Components	Mike	Done	Done	3 wk	
Messaging	Sameer	Done	Done	3 wk	
Threads Package	Sameer	No Document		1 wk	
Security Design	**Igor/Amit**	**Not Done**	**Not Done**	**TBD**	
Client Group					
Cache Prefetching	Anne	Done	Done	1 wk	[redacted]
LSM + Plug in	Anne	Done	Done	1 wk	
Client UI	Anne	Done	Done	1 wk	
Client Installer	Anne	Done	Done	1 wk	
Start Client	Anne	Done	Done	1 wk	
Application Install Mgr	Nick	Done	Done	TBD	
Piracy	Nick	Done	Done	TBD	
File Spoofer	Curt	Done	Done	1 wk	
eStream File System	Curt	Done	Done	8 wk	
NoCluster Driver	**Curt**	**Status TBD**	**Starus TBD**	**2 days**	
eStream Cache Manager	Dan	Done	Done	8 wk	
Client Network Interface	Dan	Done	Done	2 wk	

Implementation Plan

Milestones

[redacted]

ECM (RAM disk cache) and EFSD executes a local "himom" executable
Photoshop is installed locally and successfully executed from estream sets and appInstallblk produced by builder
App Server and EMS integrated to copy "himom" executable using a dummy client
App Server, EMS and CNI integrated to copy "himom" executable from "himom" estream sets
office is installed locally and successfully executed from estream sets and appInstallblk produced by builder
App Server, EMS, ENI, ECM and EFSD integrated to run "himom" from estream sets on server
Following applications built and tested with local installation

- ***Adobe Premier***
- ***Macromedia Director and Shockwave***
- ***Corel Suite***
- ***Lotus Suite***

[redacted]

Photoshop is installed by AIM and executed from estream sets on App server

- *No Subscription*
- *No License Management*
- *RAM cache for ECM*
- *Installation of Photoshop using AIM*

[redacted]

Photoshop is installed by AIM and executed from estream sets on App server

- *No Slim Server*
- *Disk based cache for ECM*

Estream includes initial prefetched pages and these pages are prefetched during installation
Fully functional estream bits (includes initial prefetched pages)
Client software is run as a service
App Server is started by Monitor
Admin UI to stop and start app Server
Application subscription from web server
installation on client after subscription

[redacted]

Testing environment is setup (configuration of 3 servers and one client)

Photoshop runs with the following additional functionality

Leads for milestone: Amit and Nick

Slim Server
http protocol
CNI supports uique message ids for NAD
Fully functional LSM
Real Accesstokens
Uninstall applications
Anti-Piracy support
AppServer and SlimServer fail-over
File spoofing

[redacted]

Clean builds by integration (George) **(Raj will drive this)**

[redacted]

Office is running with full functionality

Restruturing of client so it can be started at boot time
Performance tuning
Improve robustness
application upgrade
Crash resiliency
All software purified and memory leaks eliminated
(May be) Applets for monitoring server components

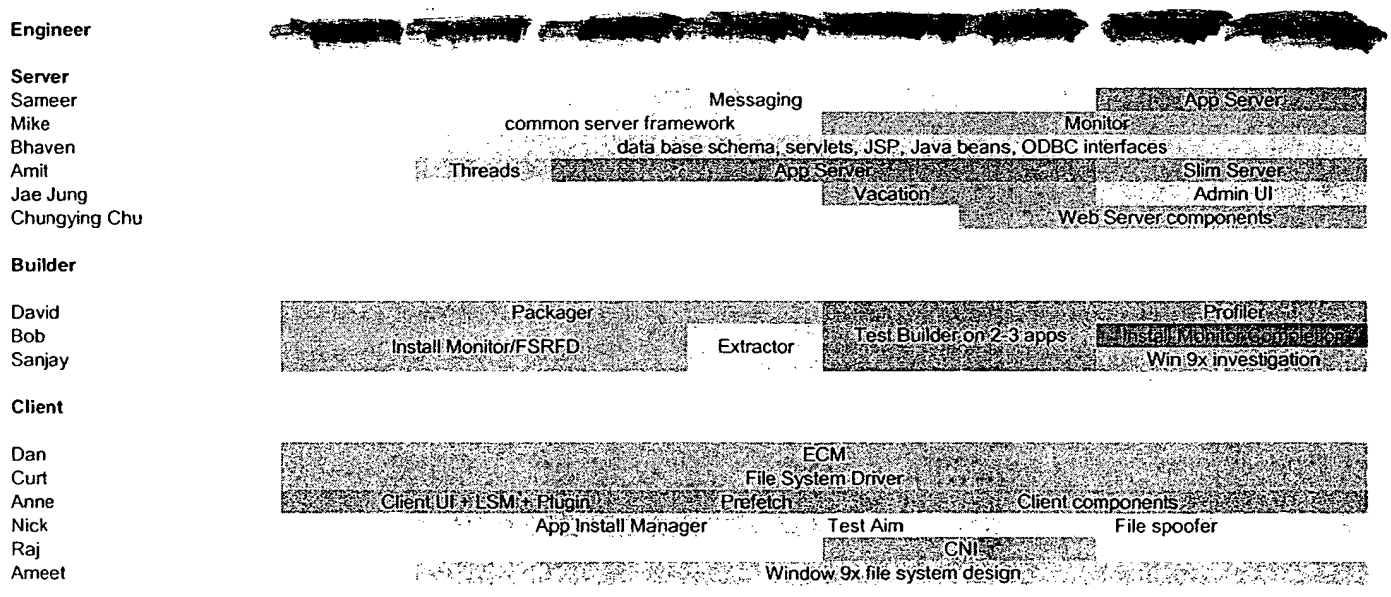
[redacted]

Office is removed from desktop of at least one person and reinstalled using estream

[redacted]

Code Freeze

[redacted]



eStream Server Component Framework
Low Level Design

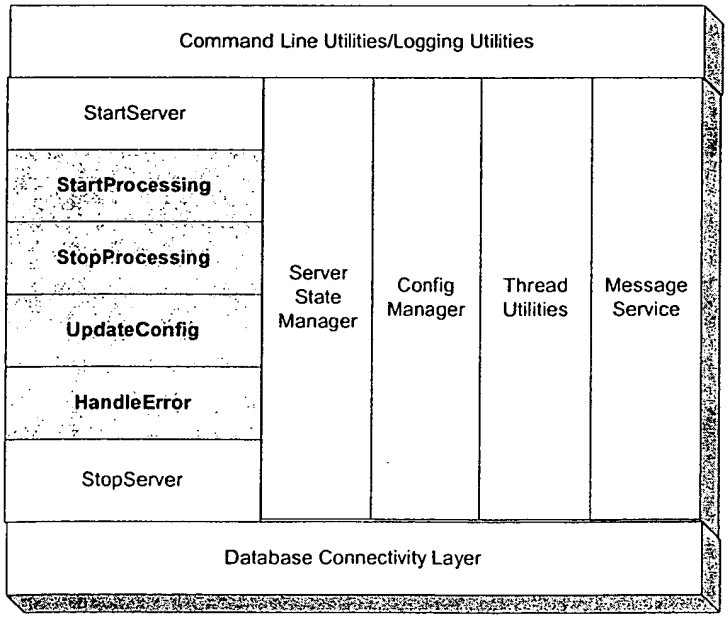
Michael Beckmann
[redacted]

Functionality

The *Server Component Framework* provides a common basis on which server components are implemented. The framework provides a number of services such as common server initialization and configuration, messaging, state management, logging, and error handling. The component framework ties together many of the core utilities provided for the server components.

The advantage of the framework is that heterogeneous server components can be managed in a consistent manner with the expectation that all server components will communicate and behave consistently within the system.

All server components with the exception of the web server will be built on top the *Server Component Framework.* To make use of the *Server Component Framework,* a specialized server component will need to extend the framework by implementing the methods high-lighted in gray. Implementing these interfaces makes the specialized server component "plug-able" within the framework.



The following table give a brief description of each of the routines that need to be specialized by each server component to make it plug-able into the Server Framework:

StartProcessing	Specialized server component routine to request the server component to start processing work.
StopProcessing	Specialized routine to request the server component stop processing work and transition into an idle state
UpdateConfig	Specialized routine to dynamically update configurations while a component is either in the processing or idle state.
HandleError	Specialized routine to handle the occurrence of an error

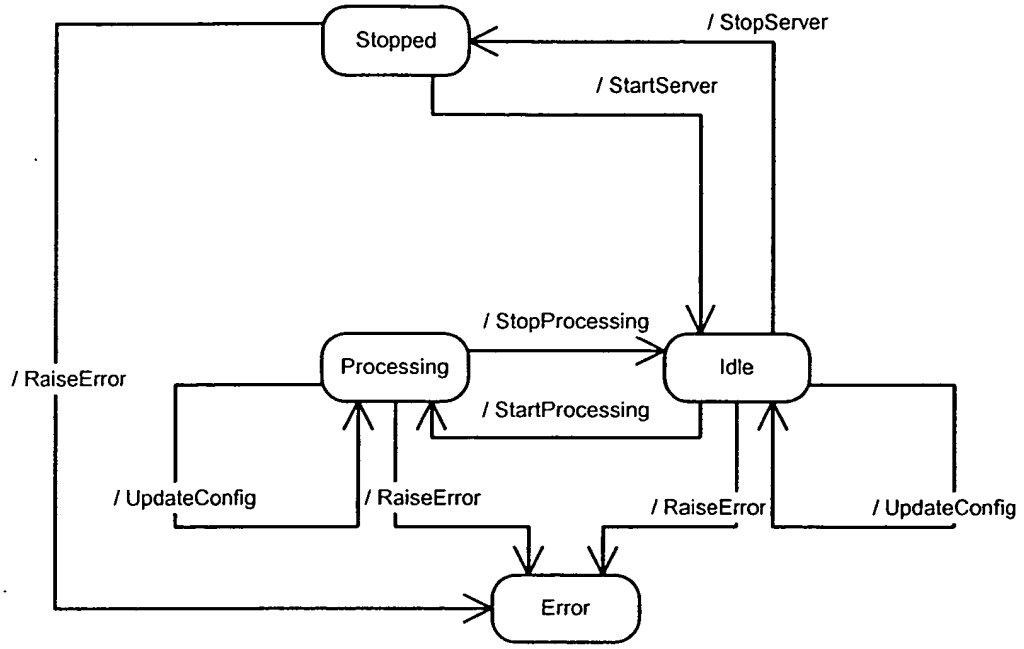
Server State Manager:

At the heart of the server component framework is the *Server State Manager*. The server state manager is a set of interfaces that initiate and manage state changes within a server component. All Server components, by virtue of being built on top of the component framework, can be managed uniformly across a deployment.

The *Server State Manager* implements a simple state machine that is shared between components. It manages the state transitions within the server component. Additionally, the state manager maintains current state information for each server component and logs state transition history in the event that a server component terminates unexpectedly.

As specified above, each server component is required to implement a number of transition methods, with pre-defined signatures, which the state manager will execute when making a state transition.

The following diagram shows the state diagram and the associated transitions:



Message Service:

The *Server Component Framework* depends on a message service which is used by the Server State Manager and Configuration Manager to communicate with the System Monitor.

The *Server State Manager* uses the messaging service to listen for state change requests from the System Monitor which it satisfies by returning the current state, any up-to-date status, and load information.

The *Configuration Manager* uses the message service to request configuration information from the *System Monitor*. Although each server component could easily go to the database for configuration information, it has been decided to go through the monitor as to save db licensing costs.

See below for more details on messaging protocols for the *Server State Manager* and the *Configuration Manager*. Also, refer to the low-level design document for details on the design of the eStream Messaging Service (EMS).

Configuration Management:

The configuration management utility is used by all server components to manage the server configurations. It provides the following functionality:

- Configuration for a server consists of a set of name – value tuples where the values themselves can be a set of name-value tuple.
- Servers can load the complete configuration from the database (indirectly).

- Servers can load the configuration for a given name.
- Servers can load the configuration from a flat file also.

On the Server Manger interface, configuration will appear as a table containing name – value tuples. The table may be hierarchical to represent nested structures containing the values which can themselves be name values. An example of a simple name-value pair would be:

port 8080

An example of nested name values would be:

Applications:
word.exe *windows2000sp3*
excel.exe *win98sp4*

On a flat file the configurations will always be name-value pairs. To represent one level nested structure the format would be:

Applications word.exe windows2000sp3
Applications excel.exe win98sp4

A common set of configurable parameters is defined for all server components. These configurations are maintained by the *Server Component Framework* in collaboration with the *Configuration Manager.* All configuration information is persistently stored within the database. The common configurations are used to initialize the server component after the component process has been launched. Refer to the configuration table below for more details on common configurations. Specialized server components can support additional configurations (non-common) depending on the server type. These configurations are read from the database and updated when a server component starts processing. They can also be updated dynamically while a server component is processing through the use of the **UpdateConfig** interface.

The list of common configurations include:

Information	**Supports Dynamic Config**	**State**	**Description**
ServerID	No		Unique identifier for server components. This server identify is unique within a deployment. This ServerID is not known to eStream clients. Its purpose is as a handle to uniquely identify server components.
ServerType	No		Identifies the type of server component. One of the following applies: ▪ Primary Monitor

			▪ Backup Monitor ▪ Application Server ▪ SLiM Server
DbUser	No		User name string required for database connectivity for this server ID
DbPasswd	No		Database password associated with the DbUser
Dsn	No		Data Source Name used to access the database.
PortNum	No		PortNumber used for light-weight messaging listener
MachineID	No		Machine ID is used to get at important machine information needed for all server components such as: ▪ IP address for the machine server component is hosted on ▪ Domain name for the machine ▪ Machines name
AutoReStart	Yes	Any State	Flag indicating that server component process can be restarted automatically without manual intervention. This info is consumed by the System Monitor.
HeartBeat-TimeOut	Yes	Any State	Specifies the timeout period for the listener. If the timeout period is reached. The component assumes that it has lost the connection. All Server components have a listener by which they receive instructions from the primary system monitor. Even the monitor has a listener that communicates with the Server Admin UI.
HeartBeatRate	Yes	Any State	Frequency at which the heart-beat is sent to this server component. Specified in milliseconds. This item is consumed by the System Monitor.

Command Line Utilities:

The *Command Line Utilities* component provides a consistent way to define and process command line arguments. To use this utility, the using component must define a table of arguments, which defines the valid set of arguments, whether or not they are required, and any default values.

Arguments are specified on the command line as name/value pairs. The utility implements the following command line syntax to support the name/value pairs. The argument syntax is defined as follows:

<name>=<value>

name	Name is an alpha-numeric identifier. The Name can be of arbitrary length as supported by the system however shorter names are recommended. Names are case sensitive
value	Any alpha-numeric value. Punctuation characters may also be used. Values are

	case sensitive

There can be no spaces between the <name>, "=", and the <value> elements. The existence of one or more spaces or tabs delineates separation between arguments on the command line.

Example: server.exe sid=267 dsn=oracle user=michaelb passwd=mypasswd

- If a named argument is specified more than once on the command line, subsequent arguments will cause a diagnostic to be issued and the argument will be ignored.
- This utility allows the user to specify default values for arguments. If a default value is defined then the argument will be processed with its default in the event that the argument is not specified on the command line.
- This utility allows the user to tag specific arguments as required. If the required argument is not specified on the command line this utility will raise a diagnostic for the required argument. Not specifying a required argument will cause a fatal error.

The following options are supported:

sid	Server Component Identifier. Each server component within a deployment is uniquely identified via the sid. The sid is a handle into the database for accessing information unique to a specific server component.
dsn	Data Source Name. A data source name is necessary to establish an ODBC connection. Data Source Names are generated by an ODBC administrative tool
dbuser	User name. For database access security, all components need to connect as a specific user.
dbpasswd	password associate with the dbuser

Logging Utilities:

All servers and clients in eStream 1.0 need to log the error and access data. Logging enables component debugging and auditing support.

EStream Framework should provide logging with the following features:

- Each component will have an error and optioanally an access log file. The names of these files would be <component>_error.log and <component>_access.log.
- The files will be located in the <eStream1.0 Root Dir>\logs directory.
- The error log files will have messages with the following priorities:
  - 4-Low : A warning which can be ignored.
  - 3-Medium: A warning which needs to be looked into.
  - 2-High: Recoverable Error in the component.
  - 1-Critical: Fatal Error. Needs admin assistance.

- Logging level should be configurable. The following levels are to be supported.
  - 0: Only errors will be logged. This will be the default level.
  - 1: Errors and Warnings to be logged.
  - 2: Errors, Warnings and Debugging information to be logged.
  - 3: Errors, Warnings and advanced Debugging (like memory dumps, tcp stack dumps etc) to be logged.
- Log Wrapping to be supported. The log files will wrap at a predefined size. On wrapping the following actions will occur:
  - Any existing <logfile>.bak will be deleted from the system.
  - The current <logfile> will be backed to <logfile>.bak
  - The component will continue logging to the <logfile>.

For each eStream client and server component logging the log files (component_error.log and component_access.log) should be written in eStream1.0Root\logs directory. The formats for the log files will be as follows:

Error Log:

[HEADER]
[TimeStamp] [Thread ID] [Priority] [Message]
...
[FOOTER]

An example of this log format would be:

```
**********************************************
Omnishift eStream Application Server
Server Started.
StartTime: [redacted]16:31:19 -0700
IP Address: 1.1.1.1
Logging Level: 3
**********************************************

[[redacted]16:31:19 -0700] 0 2-High Cannot connect to the database.
Invalid Username/Password.
[[redacted]16:31:19 -0700] 1 1-Critical Cannot start the HTTP listener
at port 80.
[[redacted]16:31:19 -0700] 0 1-Critical Shutting down the server.

**********************************************
Omnishift eStream Application Server
Server Stopped.
StopTime: [redacted]16:35:19 -0700
IP Address: 1.1.1.1
Logging Level: 3
**********************************************
```

Format of Access Log Message:

[HEADER]
[TimeStamp] [Thread ID] [Message]
[FOOTER]

Data type definitions

Server State:

The server components can be in any one of the following states:

State	Description
STOPPED	If a server is in the STOPPED state then the component process is not running.
IDLE	Server component is up and running. The server has been initialized with the common configuration and the messaging system has been enabled. The listener is actively waiting on the System Monitor for transition requests. The server component is not processing any work specific to this servers specialization.
PROCESSING	Server component is actively taking requests and processing work specific to its specialization. ie. serving access tokens, and application file requests.
ERROR	An error has occurred in the system. Unless the server component is configured with **AutoReStart** and ERROR state must be manually cleared by the server-side administrator.

Server State Transitions:

Changes in server component state are initiated either by the *System Monitor* or directly by the server-side administrator for the system monitor. The exception to this is when an error condition is raised by a server component. In this case, the component will initiate the state change itself. The following state transitions are supported:

Action	Description
START_SERVER	Server is expected to be in the STOPPED state. If a server component is configured to support AutoReStart then the ERROR state is also a valid state from which to initiate this action.
STOP_SERVER	Causes the server to exit its process. The server can be stopped from any state.

START_PROCESSING	Causes the server to change from the IDLE state to the PROCESSING state.
STOP_PROCESSING	Causes the server to change from processing to IDLE state.
UPDATE_CONFIG	Request that the server read its configuration from the configuration manager and change its configuration.
RAISE_ERROR	Request that the server go to ERROR state. This causes an error handler to be called. If the error is fatal it will cause immediate termination of the server process.

Finite State Table:

```
FSMTableEntry ServerStateMgr::FSMTable[] =
{
        { START,{{START_SERVER,STOPPED, START_SERVER,NULL},
                {START_PROCESSING, STOPPED ,START_PROCESSING, NULL},
                {NULL_REQUEST,  NULL_STATE, NULL_REQUEST, NULL}} },

        {STOPPED, {{START_SERVER, IDLE, NULL_REQUEST, &StartServer},
                {START_PROCESSING, IDLE, START_PROCESSING, &StartServer},
                {RAISE_ERROR, ERROR, NULL_REQUEST, &HandleError},
                {NULL_REQUEST,  NULL_STATE, NULL_REQUEST, NULL}} },

        {IDLE,{{START_PROCESSING, PROCESSING,  NULL_REQUEST,
&StartProcessing},
                {STOP_SERVER, STOPPED, NULL_REQUEST, &StopServer},
                {RAISE_ERROR, ERROR, NULL_REQUEST, &HandleError},
                {UPDATE_CONFIG, IDLE, NULL_REQUEST, &UpdateConfig},
                {NULL_REQUEST,  NULL_STATE, NULL_REQUEST, NULL}} },

        {PROCESSING,{{STOP_PROCESSING, IDLE, NULL_REQUEST,
&StopProcessing},
                {UPDATE_CONFIG,PROCESSING, NULL_REQUEST,
&UpdateConfig},
                {STOP_SERVER, IDLE, STOP_SERVER,  &StopProcessing},
                {RAISE_ERROR, ERROR, NULL_REQUEST, &HandleError},
                {NULL_REQUEST,  NULL_STATE, NULL_REQUEST, NULL}} },

        { ERROR,{{STOP_SERVER, STOPPED,NULL_REQUEST, NULL},
                {NULL_REQUEST,  NULL_STATE, NULL_REQUEST, NULL}} }

        {NULL_STATE,{{NULL_REQUEST, NULL_STATE, NULL_REQUEST,
        NULL} } }
};
```

Messaging Service Protocol:

A light-weight messaging protocol is needed to facilitate communication between server components. The primary purpose of the messaging protocol is to communicate transition requests to the server components. In response, server components communicate state, status, and load information back to the *System Monitor*.

The messaging protocol supports two primary message types. 1) Requests for the *System Monitor* to perform on other servers. 2) Requests to the server components themselves. These message types are distinguished through the protocol as described below. If the receiver ID and the target ID are identical then the request is for the receiver. If the target is different than the receiver, the message is for the *System Monitor* to enact a request on the target component.

All requests are required to be acknowledged. Without an acknowledgement the message is considered un-received.

OpCode	senderID	receiverID	targetID	Data

The following table describes the protocol used by the Server State Manager in its communication with the System Monitor.

OpCode	Description	Data
0x01	Request for current state	None
0x02	Acknowledgment	▪ Current state ▪ Load info ▪ Status info
0x03	Stop Server request. Acknowledged with 0x02 message	None
0x04	Start Server request. Only valid for System Monitor. Acknowledged with 0x02	None
0x05	Start Processing Request. Acknowledged with 0x02	None
0x06	Stop Processing Request. Acknowledged with 0x02	None
0x07	Update Configuration Request. This is a request for a server component to request its specialized configuration information from the System Monitor and update itself. Acknowledged with 0x02.	None

Interface definitions

Server State Manager:

```
class ServerStateMgr
{
```

```
private:
        ServerState CurrentState;
        static FSMTableEntry FSMTable[];

public:
        ServerStateMgr(void);
        ~ServerStateMgr(void);

        ServerState SetState(ServerState);
        ServerState GetState(void);
        ServerState ProcessRequest(ServerRequest);
};
```

SetState	**Description:** Sets the current state of the server component. 1. Log the state change request 2. Update the state field within the server component in memory data structures. 3. Send message to requester informing them of the successful state change. Note: **SetState** does not update the database directly as in the orginal design. The database is updated by the *System Monitor* once it has received an acknowledgement. A state transition is not complete until **SetState** returns successfully and the Monitor has update the database. **Input:** state value to set current state to. **Output:** current state after the new value has been set. If an error occurs will go to error state. **Errors:** 1. Invalid state argument 2. Failure to either connect or commit state change to the database.

GetState	**Description:** returns the current state. This function does not read from the database to get the current state. The assumption is that if the server component is up and running and that it maintains a valid state. **Input:** none. **Output:** returns the current state. **Errors:** None. Will always return a valid state.

ProcessRequest	**Description:** request to the Server State Manager to change server state. This routine implements the guts of the state machine. 1. Get the current state, and transition request 2. Index into the FSM table and continue to transition from state to state until the transition request is satisfied. 3. Each state transition calls the specialized transition routines for each component. 4. Call to **SetState** to complete each state transition. 5. In the case of an error the state machine will process a RAISE_ERROR request which will call the specialized **Han-**

	dleError and transition to the ERROR state. **Input:** server transition request. Refer to table of valid requests defined above. **Output:** current state after the request has been completed. **Errors:**

Server Component Framework:

```
class ServerComponent: ServerStateMgr{    // abstract base class
private:
        ErrorInfo*      Error; // maintains error if error was detected
        ServerConfig*   Config;  // holds common configuration
        Connection*     Listener; // messaging        utility
public:
        virtual int StartServer(void); // may be specialized by a server component
        virtual int StopServer(void); // may be specialized
        virtual int StartProcessing(void) = 0; // must be specialized
        virtual int StopProcessing(void) = 0; // must be specialized
        virtual int UpdateConfig(void) = 0; // must be specialized
        virtual int HandleError(void) = 0; // must be specialized
        void Run(Request);
}
```

StartServer	**Description:** Called by the *Server State Manager* when a server component is to be started. The **StartServer** routine is provided as part of the SeverComponent class. It performs the following: 1. Send request to System Monitor to request an update of common configuration information. 2. Apply the configuration information to the server component. 3. Construct a listener connection object and start the message service. 4. Return success or failure. Note: ▪ This routine must return immediately to the main thread. Otherwise the *Server State Manager* will be blocked. ▪ Successful return from the **StartServer** routine will put the server into the IDLE state. **Input:** None. **Output:** Value of 0 if successful else error condition **Errors:** May return negative error condition

StopServer	**Description:** Called by the *Server State Manager*. 1. Perform any necessary cleanup. 2. Send last acknowledgment confirming shutdown to requester 3. Shut down the messaging system and the listener. 4. exit process Note: The monitor will update the database and perform logging.

	Input: None. **Output:** Value of 0 if successful else error condition **Errors:** May return negative error value

StartProcessing	**Description:** Called by the *Server State Manager*. This routine must be defined by each specialized server component. This routine is used to provide all functionality unique to different types of servers. 1. Spawn a primary processing thread (also known as the boss thread). a. Read server specific configurations unique to this type of server component from the System Monitor b. Spawn worker threads. Depending on the server type this routine does the heavy lifting to either process access tokens and renewals in the case of SLiM server, or process file requests for application servers, or manage and monitor the server components in the case of the *System Monitor*. Note: ▪ This routine must return immediately so that the *Server State Manager* can continue to operate in the main thread. ▪ This routine may make use of the *Server Configuration Manager* for obtaining specialized configuration information **Input:** None **Output:** Value of 0 if successful else error condition. **Errors:** TBD

StopProcessing	**Description:** Called by the *Server State Manager*. This routine must be defined by the specialized server component type. 1. Reverse all actions performed by the **StartProcessing** routine. All worker threads should be joined or pooled in waiting state. Successful return from this routine will put the server component into the IDLE state. **Input:** None. **Output:** Value of 0 if successful else error condition. **Errors:** TBD

UpdateConfig	**Description:** Called by the *Server State Manager*. This routine must be defined by the specific server component type. The purpose of this routine is apply dynamic configurations or update specialized configurations that are unique to this server component. <may require adding a new state to separate dynamic and static configurations> **Input:** None. **Output:** Value of 0 if successful else error condition. **Errors:** TBD

HandleError	**Description:** Component defined error handling routine to handle errors such as timeouts, etc. This routine will need to handle a number of error cases as are possible by the specialized component. The error information is maintained with the ServerComponent class. **Input:** None. **Output:** Integer value designating a handled error or failure. If the error cannot be handled then it is fatal. **Errors:** TBD

Run	**Description:** This routine implements the main processing loop for the server component and runs in the main thread. This routine drives the server component by initiating state requests from the *System Monitor.* Note: The *Server State Manager* always runs in the main thread. 1. Call **ProcessRequest** to transition the server component into the initially requested state. 2. Enter main processing loop a. Check for requests from the message service. b. Call **ProcessRequest** to service the request. c. Send acknowledgement for the request to the message service. Acknowledgement includes new state, load info, and status. **Input:** Initial Transition Request **Output:** None. This routine should never return **Errors:** None.

Server Component Main Loop:

The following main loop is common to all server components:

```
void ServerComponent::Run(ServerRequest Request)
{
        ProcessRequest(Request);
        while (1)
        {
                Request = Listener->GetRequest();
                ProcessRequest(Request);
                Listener->AckRequest(Request, GetState, GetLoad, GetStatus);

        }
}
```

```
#include "ServerArgs.h"
#include "Server.h"
```

```
int main(int argc, char* argv[]) {
        Args = new ArgList();
        Args->ProcessArgList(argv, argc);
        Server = new ServerComponent(GetValue(SID),
                                     GetValue(DSN),
                                     GetValue(DBUSER),
                                     GetValue(PASSWD));
        Server->Run(START_PROCESSING);
}
```

Command Line Utilities:

```
class NameValuePair
{
        private:
                char* Name;
                char* Value;
        public:
                NameValuePair();
                ~NameValuePair();
                char* GetValue(void);
                char* GetName(void);
                char* SetName(char*);
                char*SetValue(char*);
};
```

```
typedef int (*pFunc)(NameValuePair*);

struct ArgTblEntry
{
        char*   Name;
        bool    Required;
        char*   DefaultValue;
        pFunc   ProcessFunction;
};
```

```
ArgTblEntry const ServerArgsTbl[] = {
        {"sid",         true,   0,      &ProcessSId},
        {"dsn",         true,   0,      &ProcessDsn},
        {"dbuser",      true,   0,      &ProcessDbUser},
        {"dbpasswd",    true,   0,      &ProcessDbPasswd},
        {0,             0,      0,      0}
};
```

```
typedef vector<NameValuePair*>    ArgVector;
```

```
class ArgList
{
        private:
                ArgVector                   ArgVec;
                const ArgTblEntry*          ArgTbl;

        private:
                NameValuePair*              ParseArg(char* Arg);
                char*                       ParseName(char* Arg);
                char*                       ParseValue(char* Arg);
                int                         ProcessArg(NameValuePair*);
                int                         FinalizeArgs(void);

        public:
                                            ArgList(const ArgTblEntry*);
                int                         ProcessArgList(char* argv[], int argc);
};
```

ProcessArgList	**Description:** Process the entire argument list. In a loop for each argument argv[] ... 1. Call **ParseArg** passing in argv[]. 2. **ParseArg** passes the result to **ProcessArg** 3. After processing the entire argument list and exiting the loop call **FinalizeArgs** **Input:** argv and argc as passed into main() entry point **Output:** integer value designating success or failure **Error:**
ParseArg	**Description:** Takes a char* argument and verifies that it follows that name/value syntax defined as <name>=<value> **Input:** Next char* argument on the list **Output:** **NameValuePair**. NULL will be returned in the event of a syntax error **Error:**
ProcessArg	**Description:** This routine performs the semantic analysis of an argument. 1. Look up name in the **ArgTbl** 2. Verify that the value is valid 3. Add the name value pair to a list of processed arguments called **ArgVec** list. 4. If this name value pair already exists in the list then issue a diagnostic. 5. Call the supplied processing function for this argument as specified in the **ArgTbl** **Input:** **NameValuePair** **Output:** Interger value designating success or failure (0 for success, positive integer for other errors) **Error:**
ParseName	**Description:** Verify that the Name part of the argument conforms to being alpha-numeric **Input:** char* Name part of argument **Output:** char* Name else NULL **Error:** None
ParseValue	**Description:** Verify that the Value part of the argument conforms to being alpha-numeric and/or punctuation characters **Input:** char* Value part of argument **Output:** char* Value else NULL **Error:** None
FinalizeArgs	**Description:** Post process the argument list. The purpose of this routine is to validate that all required arguments have been defined on the command line. Also processes and adds default arguments to the **ArgVec**. **Input:** None **Output:** Success or Failure **Error:**

Configuration Manager:

```
class Tuple {
        string name;
        Value value;
};

class Value {
        int type;
};

class StringValue: public Value{
        string value;
};

class TupleValue: public Value {
        vector <tuple> tupleArray;
};

typedef vector < tuple > ConfigArray;

class ServerConfig {
private:
        ConfigArray Array;
public:
        ServerConfig(serverId, dsn, dbuser, dbpasswd); // Initialize from db
        ServerConfig(serverId, string filename); // To initialize from a file.

        ConfigArray* GetConfigArray(int serverId);
        Tuple* FindConfig(string Name);
        int Reload(void);
        Tuple* GetConfig(int serverId ,string Name);
};
```

ServerConfig	**Description:** Constructor for Configuration Manager. 1. Initializes configuration manager. 2. Opens the database and gets configuration array **Input:** Server Id, Data Source Name, Database User name, and database users password. **Output:** None **Errors:**
ServerConfig	**Description:** Constructor for Configuration Manager. 1. Initializes Configuration Manager. 2. Opens configuration file and reads configuration array. **Input:** filename of flat-file configuration. **Output:** None **Errors:**

GetConfigArray	**Description:** Returns the entire configuration for a given server id. This routine always retrieves its information either from the flat file or the database. **Input:** ServerId specifying which server to retrieve configuration for **Output:** Returns a vector holding the configuration or NULL **Errors:**

GetConfig	**Description:** Returns the configuration for the specified name. This routine always retrieves its information either from the flat file or the database. **Input:** ServerId specifying the server to retrieve configuration for and Name of configuration item. **Output:** Configuration Tuple. A Tuple may be a nested Tuple. NULL if an error is encountered. **Errors:**

FindConfig	**Description:** Returns the Tuple specified by the name. This routine does not go to the database or flat-file to get its value. Rather it finds the value in the ConfigArray maintained by the Configuration Manager. **Input:** Name of the configuration item. **Output:** Configuration Tuple. NULL if an error is encountered or the Tuple does not exist in the current configuration. **Errors:**

Reload	**Description:** Reloads the entire configuration from the database or flat-file. This routine may reload its configuration indirectly through the use of the System Monitor. In this case it will make a message request to the monitor and listen for the configuration results. **Input:** None **Output:** integer specifying success or failure. Zero will be returned in the case of Success. A negative value in case of error. **Errors:**

Logging Utilities:

```
class LogManager
{
private:
        char* FileName;
        int MaxFileSize;
        char* ResourceFile; // message catalog file

        char* GetMessage(MsgNum, MsgStr)
public:
        LogManager(ServerId,Size=10);
        LogMessage(MsgStr);
        LogMessage(ThreadId, MsgNum, MsgStr, ...);
```

```
};
```

LogMessage	**Description:** Write message out to log file. There are two forms of LogMessage. The first will write out a message buffer as is (unformatted) bypassing the resource file. The second form will format the message. Both forms of LogMessage always pre-append a time stamp. 1. Lookup message number in the resource file and get message string 2. format the log message using time stamp, thread id, etc. 3. write out message into the log file. **Input:** Thread Id, Message Number, Message String, and variable number of arguments. **Output:** None. **Error:**

GetMessage	**Description:** Routine returns a message string from the resource file for the message number specified. **Input:** Message number, C Locale text string. **Output:** Message string. Either way, Get Message will always pass a return a valid message string by either returning the string from the resource file or by passing back the MsgStr passed in. **Error:** If an error occurs trying to get a message from the resource file, a message will be logged to the error log.

```
class ErrorLog: protected LogManager
{
private:
        LogLevel ErrorLogLevel;
public:
        ErrorLog(ServerId, LogLevel=0, Size=10);
        LogError(ThreadId, ErrorNum, ErrorMsgStr, ...);
};
```

LogError	**Description:** Writes output to error log file. 1. Check that the message level against the current ErrorLogLevel. 2. Format the message and call the long form of **LogMessage** to write the buffer out to the file. **Input:** 1. ThreadId: Thread identifier to help with the debugging process. 2. ErrorNum: Error number used to uniquely identify an error message in the resource file. 3. ErrorMsgStr: Message string which includes stdio like string formatting. 4. ...: variable list of arguments to be inserted into the message string per the format. **Output:** None. **Error:**

Testing design

Each of the components that make up the Server Component Framework will be able to be tested independently of the other components. Each component will have a main entry point defined within a testing .exe to accomplish the Unit testing phase.

Testing of the component framework will be done in phases. Each of the phases is described below along with its dependencies.

Phase 1: Unit testing Test basic components that make up the framework. Each components functionality, restrictions, and boundary conditions will be tested. Will allow testing common configurations for a single server component. This round of unit testing will test the integrated component utilities and framework.	**Dependencies:** 1. ServerComponent class 2. ServerStateMgr class 3. ArgList class 4. Logging Utilities 5. Configuration Manager (flat-file)
Phase 2: Unit testing (full functionality) Test full functionality including messaging interfaces and database connectivity.	**Dependencies:** 1. Phase 1 2. Database connectivity 3. Messaging Service
Phase 3: Integration Testing	**Dependencies:** 1. Phase 2 2. System Monitor (including backup) 3. SLiM Server, App Server, Web-Server
Phase 4: Stress Testing See section on stress testing for details	**Dependencies:** 1. Phase 3

Unit testing plans

Command Line Utilities

The Command line utilities will be tested in a stand-alone module called cmdline.exe. It will support the command line arguments defined in this document.

Configuration Manager

The configuration module is a stand-alone module which will be tested using a config-test.exe executable. The executable will exercise all of the interfaces described above. The configtest.exe executable should be testable in the DB and the non-DB mode.

Logging Utilities

The logging utility will be built as a DLL (otlog.dll). We will provide a binary otlog-test.exe which will exercise each of the interfaces mentioned above.

Server State Manager

The Server State Manager and the Server Component Framework will be tested independently of specialized components. The routines that require specialization (**StartProcessing**, **StopProcessing**, **HandleError** and **UpdateConfig**) will be provided to simply return successfully.

Stress testing plans

Stress testing will require having at least the System Monitor functionality implemented since it is used to drive the server components.

1. Test to repeatedly start, stop, reconfigure the server component.
2. Test to crash machines with server components to validate:
    a. data persistence.
    b. detection capabilities and response.
    c. auto restart.
3. Test to kill individual server component processes.
    a. data persistence.
    b. detection capabilities and response.
    c. auto restart.
4. Test lost database connectivity
5. Test lost of messaging capabilities
    a. repeatedly losing and re-establishing messaging connectivity
6. Test error recovery under adverse conditions.
7. Test recovery from running out of memory, thread resources.
8. Test recovery from threads dying.
9. etc.

Coverage testing plans

1. Goal: 100% path flow coverage. Only exceptions for known error conditions that cannot be practically reached (e.g. thread synchronization, etc.)

Cross-component testing plans

The following pair-wise testing will be performed:

1. framework/database (phase 2)
2. framework/messaging (phase 2)
3. framework (System Monitor) /framework (backup Monitor) (phase 3)
4. framework/Web Server (phase 3)
5. framework (System Monitor) /framework (Other Servers) (phase 3)

Upgrading/Supportability/Deployment design

1. Each error condition will be documented with explanations and practical work-arounds
2. Component framework will support enhanced debug option to dump additional debugging information to special log files.

Open Issues

eStream Set Format Low Level Design

Sanjay Pujare and David Lin
Version 0.7
[redacted]

Functionality

The eStream Set is a data set associated with an application suitable for streaming over the network. The eStream Set is generated by the eStream Builder program. This program converts locally installable applications into the eStream Set. This document describes the format of the eStream Set.

Note: Fields greater than a single byte is stored in little-endian format. The eStream Set file size is limited to 2^64 bytes. The files in the CAF section are laid out in the same order as its corresponding entries in the SOFT table.

Data type definitions

The format of the eStream Set consists of 4 sections: header, Root Version Table (RVT), Size Offset File Table (SOFT), and Concatenation Application File (CAF) sections.

1. Header section

- **MagicNumber [4 bytes]:** Magic number identifying the file content with the eStream Set
- **ESSVersion [4 bytes]:** Version number of the eStream Set format.
- **AppID [16 bytes]:** A unique application ID for this application. This field must match the AppID located in the AppInstallBlock. Guidgen is used to create this identifier.
- **Flags [4 bytes]:** Flags pertaining to EStreamSet
- **Reserved [32 bytes]:** Reserved spaces for future.

- **RVToffset [8 bytes]:** Byte offset into the start of the RVT section.
- **RVTsize [8 bytes]:** Byte size of the RVT section.
- **SOFToffset [8 bytes]:** Byte offset into the start of the SOFT section.
- **SOFTsize [8 bytes]:** Byte size of the SOFT section.
- **CAFoffset [8 bytes]:** Byte offset into the start of the CAF section.
- **CAFsize [8 bytes]:** Byte size of the CAF section.

- **VendorNameIsAnsi [1 byte]:** 0 if the vendor name is in Unicode format. 1 if the vendor name is in ANSI format.
- **VendorNameLength [4 bytes]:** Byte length of the vendor name.
- **VendorName [X bytes]:** Name of the software vendor who created this application. I.e. "Microsoft". Null-terminated.

- **AppBaseNameIsAnsi [1 byte]:** 0 if the vendor name is in Unicode format. 1 if the vendor name is in ANSI format.
- **AppBaseNameLength [4 bytes]:** Byte length of the application base name.
- **AppBaseName [X bytes]:** Base name of the application. I.e. "Word 2000". Null-terminated.
- **MessageIsAnsi [1 byte]:** 0 if the vendor name is in Unicode format. 1 if the vendor name is in ANSI format.
- **MessageLength [4 bytes]:** Byte length of the message text.
- **Message [X bytes]:** Message text. Null-terminated.

2. Root Version Table (RVT) section

The Root version entries are ordered in a decreasing value according to their file numbers. The Builder generates unique file numbers within each eStream Set in a monotonically increasing value. So larger root file number implies later versions of the same application. The latest root version is located at the top of the section to allow the eStream Server easy access to the data associated with the latest root version.

- **NumberEntries [4 bytes]:** Number of patch versions contained in this eStream Set. The number indicates the number of entries in the Root Version Table (RVT).

Root Version structure: (variable number of entries)

- **VersionNumber [4 bytes]:** Version number of the root directory.
- **FileNumber [4 bytes]:** File number of the root directory.
- **VersionNameIsAnsi [1 byte]:** 0 if the vendor name is in Unicode format. 1 if the vendor name is in ANSI format.
- **VersionNameLength [4 bytes]:** Byte length of the version name
- **VersionName [X bytes]:** Application version name. I.e. "SP 1".
- **Metadata [32 bytes]:** See eStream FS Directory for format of the metadata.

3. Size Offset File Table (SOFT) section

The SOFT table contains information to locate specific files in the CAF section. The entries are ordered according to the file number starting from 0 to NumberFiles-1. The start of the SOFT table is aligned to 8 bytes boundary for faster access.

SOFT entry structure: (variable number of entries)

- **Offset [8 bytes]:** Byte offset into CAF of the start of this file.

- **Size [8 bytes]:** Byte size of this file. The file is located from address Offset to Offset+Size.

4. Concatenation Application File (CAF) section

CAF is a concatenation of all file or directory data into a single data structure. Each piece of data can be a regular file, an AppInstallBlock, an eStream FS directory file, or an icon file.

a. Regular Files

- **FileData [X bytes]:** Content of a regular file

b. AppInstallBlock (See AppInstallBlock document for detail format)

A simplified description of the AppInstallBlock is listed here. For exact detail of the individual fields in the AppInstallBlock, please see AppInstallBlock Low-Level Design document.

- **Header section [X bytes]:** Header for AppInstallBlock containing information to identify this AppInstallBlock.
- **Files section [X bytes]:** Section containing file to be copied or spoofed.
- **AddVariable section [X bytes]:** Section containing system variables to be added.
- **RemoveVariable section [X bytes]:** Section containing system variables to be removed.
- **Prefetch section [X bytes]:** Section containing pointers to files to be prefetched to the client.
- **Profile section [X bytes]:** Section containing profile data. (not used in eStream 1.0)
- **Comment section [X bytes]:** Section containing comments about AppInstallBlock.
- **Code section [X bytes]:** Section containing application-specific code needed to prepare local machine for streaming this application
- **LicenseAgreement section [X bytes]:** Section containing licensing agreement message.

c. EStream Directory

An eStream Directory contains information about the subdirectories and files located within this directory. The information includes file number, names, and metadata associated with the files.

- **MagicNumber [4 bytes]:** Magic number for eStream directory file.
- **ParentFileID [16+4 bytes]:** AppID+FileNumber of the parent directory. AppID is set to 0 if the directory is the root.

- **SelfFileID [16+4 bytes]:** AppID+FileNumber of this directory.
- **NumFiles [4 bytes]:** Number of files in the directory.
- **NumEntries [4 bytes]:** Number of entries in the directory. Some entries are used for storing long file names and some are unused due to deleted files. So the NumEntries must be equal or less than NumFiles.

Fixed length entry for each file in the directory consists of 2 formats (short format for storing files with name that fit the 8.3 convention; and long format for storing long file names). Each entry is 84 bytes and the entry are aligned on every 4K page boundry. Thus, in the first 4K page of the directory, the padding consists of 12 unused bytes (52 bytes for header + 48 entries * 84 bytes per entry + 12 unused bytes = 4096 bytes). In all subsequent pages, the padding is 64 bytes (48 entries * 84 bytes per entry + 64 unused bytes = 4096 bytes):

Short Filename entry:

- **Format [1 byte]:** Format of this entry, should be 's' for short format, 'l' for long filename format, or possibly 'u', for unused.
- **ShortLen [1 byte]:** Length of short file name.
- **LongLen [1 byte]:** Length of long file name.
- **UNUSED [1 byte]:** Padding
- **NameHash [4 bytes]:** Hash value of the short file name. Algorithm TBD.
- **ShortName [24 bytes]:** 8.3 short file name in unicode
- **FileID [16+4 bytes]:** AppID+FileNumber of each file in this directory.
- **Metadata [32 bytes]:** The metadata consists of file **byte size** (8 bytes), file **creation time** (8 bytes), file **modified time** (8 bytes), **attribute flags** (4 bytes), **eStream flags** (4 bytes). The bits of the **attribute flags** have the following meaning:
  - **Bit 0:** Read-only – Set if file is read-only
  - **Bit 1:** Hidden – Set if file is hidden from user
  - **Bit 2:** Directory – Set if the file is an eStream Directory
  - **Bit 3:** Archive – Set if the file is an archive
  - **Bit 4:** Normal – Set if the file is normal
  - **Bit 5:** System – Set if the file is a system file
  - **Bit 6:** Temporary – Set if the file is temporary

  The bits of the **eStream flags** have the following meaning:
  - **Bit 0:** ForceUpgrade – Used only on root file. Set if client is forced to upgrade to this particular version if the current root version on the client is older.
  - **Bit 1:** RequireAccessToken – Set if file require access token before client can read it.
  - **Bit 2:** Read-only – Set if the file is read-only

Long Filename entry:

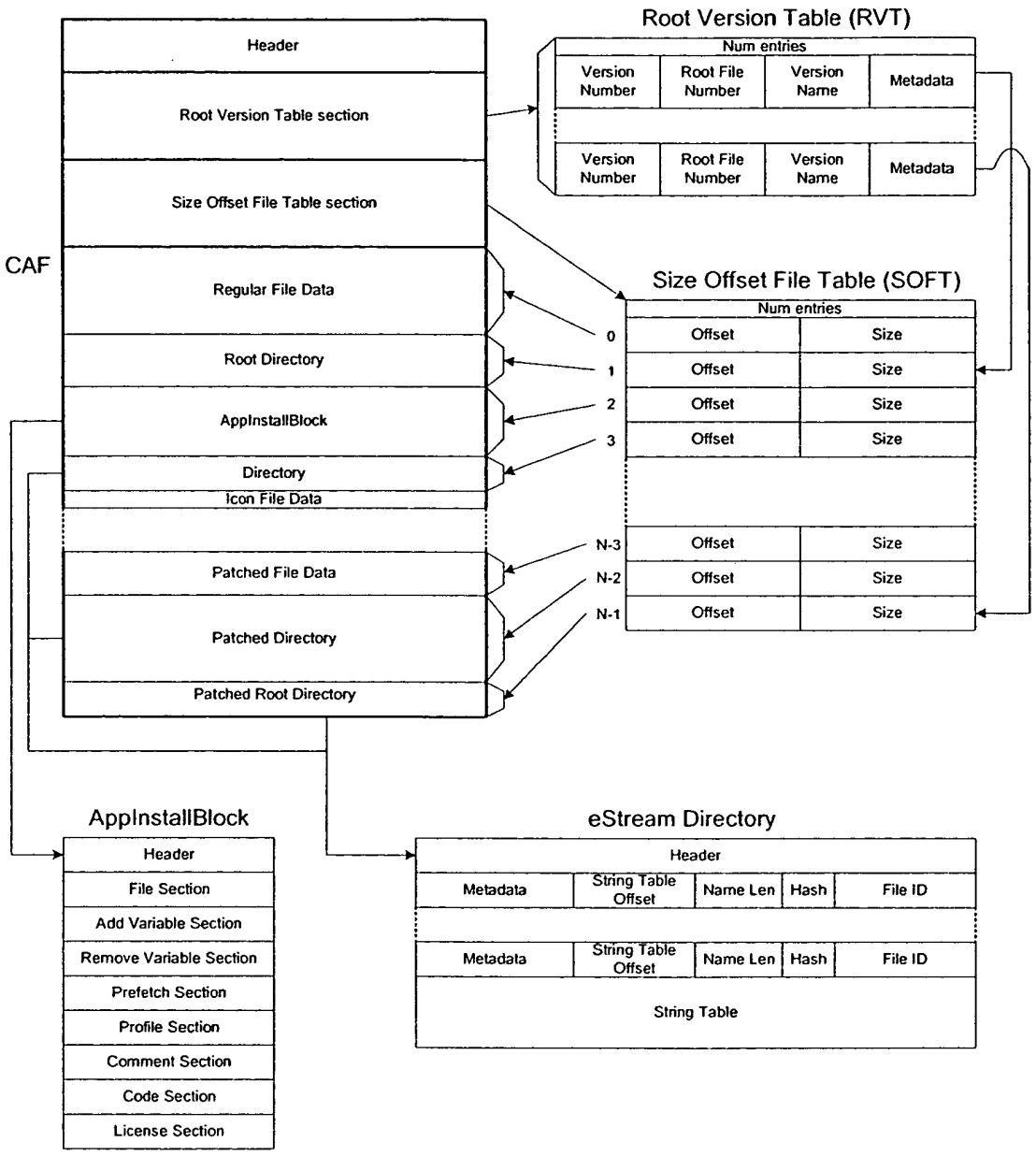
- **Format [1 byte]:** Format of this entry, should be 'l' for long filename format.

- **Index [1 byte]:** Number of this entry out of those used for this file's long name.
- **UNUSED [2 byte]:** Padding
- **NameHash [4 bytes]:** Hash value of the long file name. Algorithm TBD.
- **LongName [76 bytes]:** Long filename in Unicode format.

d. Icon files

- **IconFileData [X bytes]:** Content of an icon file.

Format of the eStream Set



v 0.2

Open Issues

- Where is the metadata associated with the Root directory located? Currently, root metadata is located in the root version table. All other files and directory metadata can be found in their parent directory.

eStream 1.0 High Level Design

Version 1.0

[redacted]

Introduction

This document describes the high level design for the eStream 1.0 product. It is essentially a summary and a tying together of the low level designs for each component in the system. The organization of this document is:

- Basic overview of the entire system
- Block diagrams for the client, server, and builder portions, showing all major components
- General discussion of each component, and pointers to the low level documents for these components
- A list of known issues

To understand the problem being solved in this design, see the "eStream Requirements Document" for information.

Note that this design is for a Windows NT4.0 and Windows 2000 client **only**. As work progresses on a Windows 95/98 client, the designs here will be updated.

Overview

eStream 1.0 encompasses the following basic features:

1. A distributed file system for application files, residing on a server and cached on a client.
2. A small client "player" program to allow local execution of applications that reside on the servers.
3. Authentication using tokens supplied by a license server to each active client.
4. A managed database of information about applications available to client machines, and subscription and usage data for each registered user.
5. Integration with service provider web servers to allow users to subscribe to apps and manage their accounts.
6. Monitoring of all servers to detect problems and allow automatic failover.
7. A build system that analyzes applications and enables them to be executed by the client and server.
8. Anti-piracy features to discourage unauthorized copying and use of subscribed applications.

As a way of overview, here are the processes that take place to enable and execute a Windows application from a client machine.

- The eStream builder is used to create an *eStream set* for the application. The application is installed on a clean machine, with the builder tools running. These will monitor all file installs and registry updates required to run the application, and encode them into a binary file—the eStreamSet—that will be installed on a service provider's eStream application server (app server).
- A user must download and install the eStream client (ECE) onto her machine, and register as a valid user from a service provider; this will be done using the service provider's web site.
- The user will subscribe to an application from the service provider; a browser module on the client machine will be notified and send a message to the ECE about this event.
- The ECE will communicate with the service provider's eStream license server (Slim server) to verify the newly subscribed app and all permissions, and will install a small portion of the application onto the client system—essentially, the registry entries, shortcuts, and small shared files necessary for execution.
- All application files that are not installed on the client will be accessed via a separate eStream file system (EFSD)
- The user will now see standard shortcuts for subscribed applications, exactly as though the app were installed locally.
- Starting an application, via a command line or double-clicking a shortcut, will cause the client machine to start executing the application on the EFSD. This means the virtual memory manager will request pages from the EFSD during page faults.
- These requests will be forwarded from the EFSD to the eStream cache manager (ECM), a component of the ECE, and on to the app server, assuming the page requested is not in the cache.
- Before any page request is fulfilled by the ECE, the client license subscription manager (LSM) will check that the user has permission to run the application, requesting an *access token* from the Slim server if an existing one has expired.
- This valid access token is sent from the client to the app server for every page request; this authenticates the request.
- The server monitor will be continually checking the state of the app servers and Slim servers. If any are down, it will take them off line.
- The client has a list of valid Slim and app servers for each registered service provider and subscribed application. If response time for any of these is bad, it will stop using it and fall back on the rest.

Block diagrams

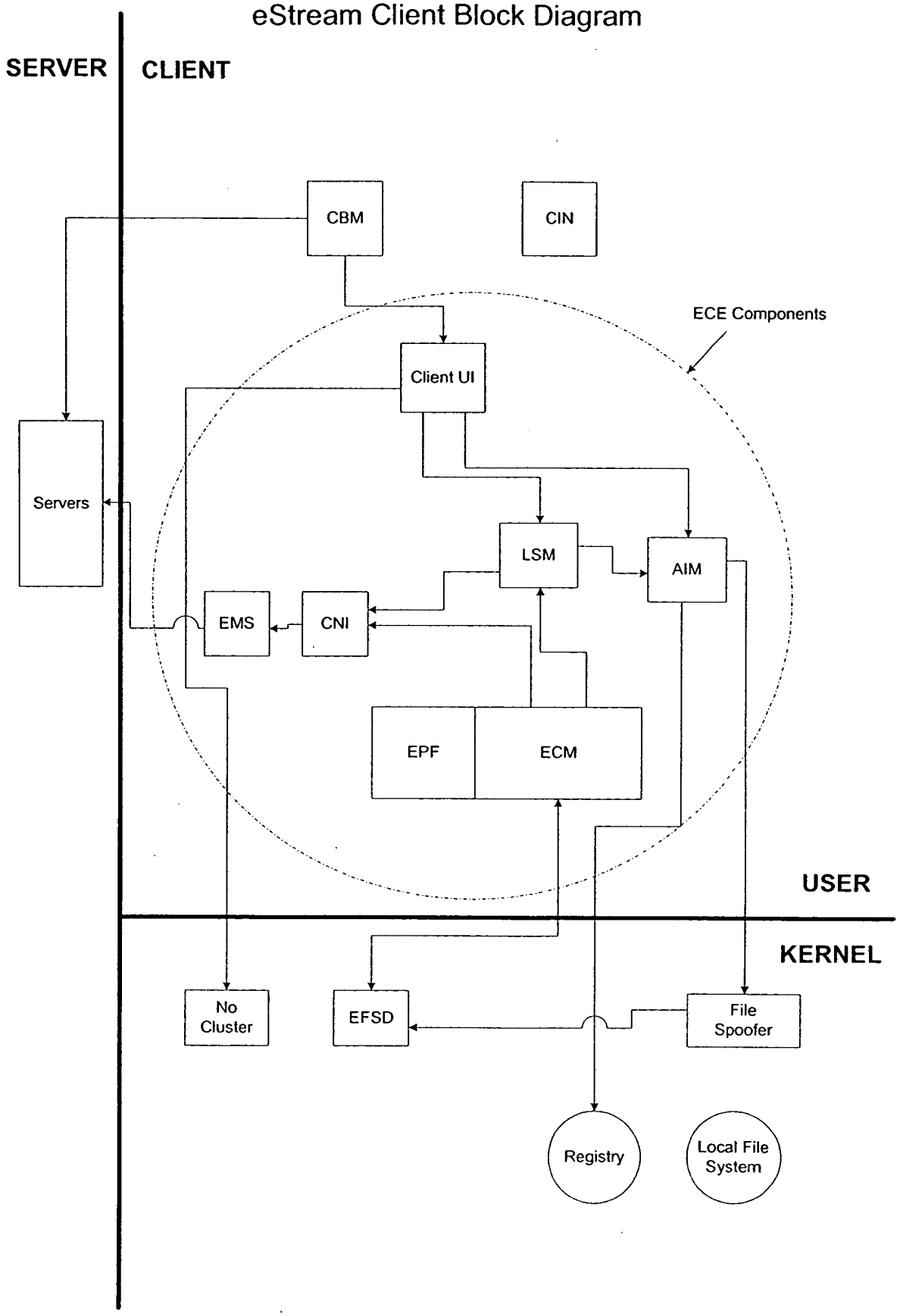
The following are simple block diagrams of the client and server components. Some conventions:

- A box represents a **logical eStream component**. A component may exist as a distinct process, or it may be grouped with other components into a common process.

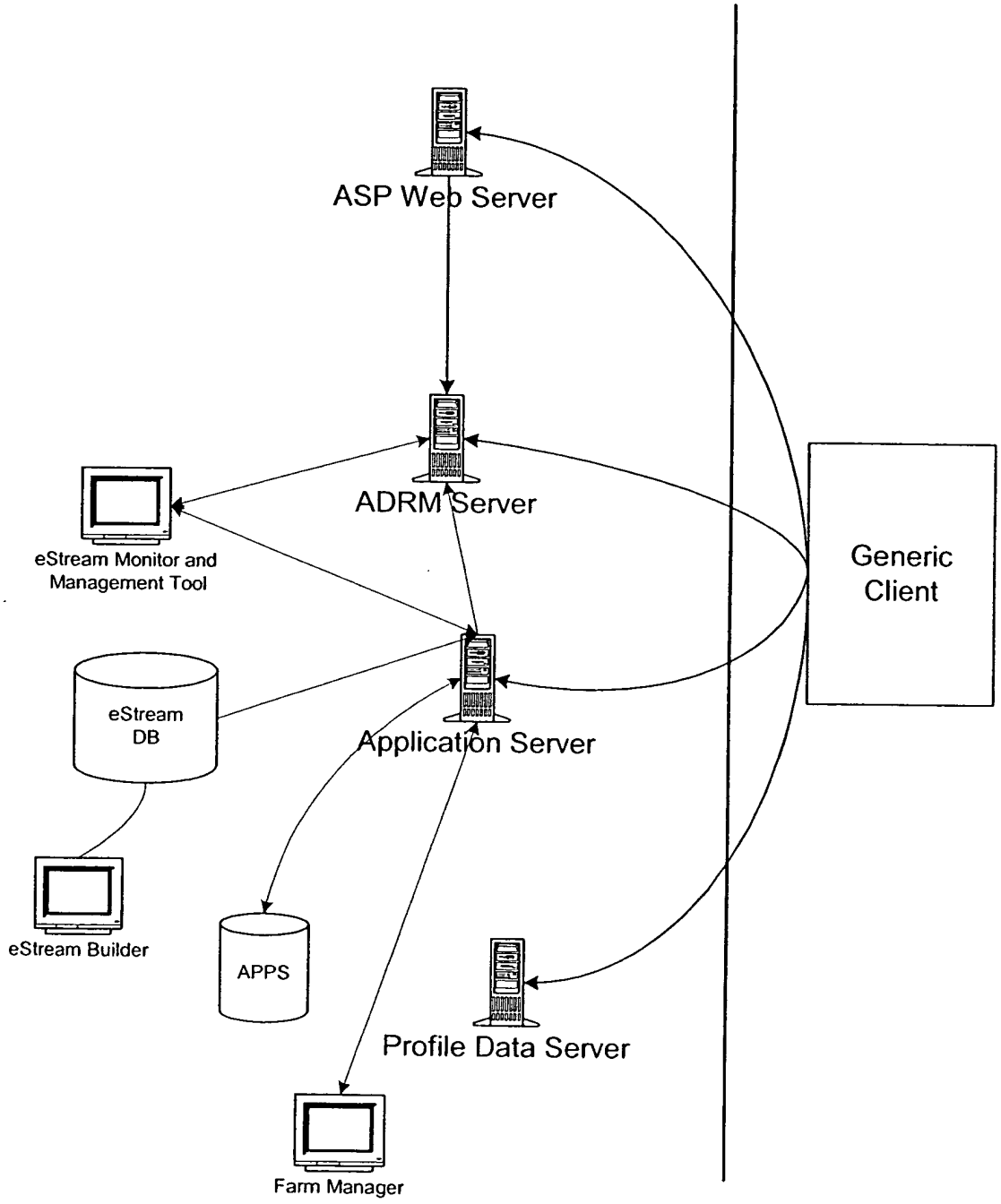
- A line between components represents an interface call from one to another. If A calls B, there's a arrow on the end of the line at B. If A and B call each other, there's an arrow on both ends of the line.

Note that data stores are **not** represented in these diagrams; if a data store is centrally managed, then there is a component that has interfaces to allow access to these data.

eStream Client Block Diagram



eStream Server Block Diagram



eStream Builder Block Diagram

???

Component descriptions

Client components

The eStream client consists of the following components illustrated in the diagram above:

- ECE: the eStream Client Executable. This is the aggregation of several user space components into a single executable, operating as a Windows service.
- LSM: the License Subscription Manager (part of the ECE). This tracks and handles all required user information needed by the client: service providers, subscriptions, and access rights.
- AIM: the App Install Manager (part of the ECE). This is responsible for installing all necessary bits onto a client machine in order to run a subscribed application. It also uninstalls all local app bits when unsubscribing.
- ECM: the eStream cache manager (part of the ECE). This is the user-space component that handles requests from the EFSD, and manages the on-disk and in-memory cache of file contents.
- EPF: the eStream PreFetch component (part of the ECE). This works closely with the ECM to handle prefetches of pages for running eStream applications (as opposed to demand fetches, handled by the ECM).
- CNI: the Client Network Interface (part of the ECE). This manages queues of requests from various client components to the app and Slim servers.
- EMS: the eStream Message Service (part of the ECE). This library, used in both the client and servers, handles the actual network sends and receives between remote machines.
- CBM: the Client Browser Module. This is a client-side web browser plugin that is used to handle notification from a service provider's web server to the ECE, when user updates have taken place.
- CIN: the Client Installer module. This small component installs, upgrades, and uninstalls all the required client software.
- FSP: the File Spoofer. This is a kernel-mode driver that is used to redirect requests, intended for local filesystems, to the EFSD. It is a file system filter driver that sniffs all Create requests to the necessary local FSDs, compares the filenames with a list of files that must be spoofed, and if a match is seen, redirects the request to the EFSD.
- EFSD: the eStream File System Driver. This is a standard Windows NT FSD, handling all necessary FS requests from the I/O Manager. It ultimately sends these requests to the ECM to be satisfied (either locally or remotely).
- No Cluster. This is a kernel-mode driver that simply disables file system page clustering for threads running as part of eStreamed applications.

ECE

The ECE is the Windows service that comprises the bulk of the user-space eStream client software. It provides an overall main program loop, as well as the user interface component for all client components that must communicate with a user.

LSM

The LSM tracks current subscription information and determines the need for license validation. It is informed of subscription changes from the client UI, and is queried by the ECM to validate accessibility to different applications, based on the license model for the subscription to that application.

The LSM has a few major tasks:

1. Keep track of what subscriptions the current user has available from all ASPs
2. Determine which application a given file is a part of
3. Acquire an access token to validate a license for file requests that require one

There are two ways that the LSM updates its list of known subscribed applications:

1. It may be informed of new subscriptions, or of applications that are unsubscribed, by the client UI, as part of a browser plugin in conjunction with an ASPs web site.
2. It may asynchronously poll an ASPs Slim servers to get updated lists of subscribed apps.

AIM

The AIM is the contact point for installation and uninstallation of applications on a client machine. It gets the requests from the LSM to install applications when the user subscribes to them, and it gets requests from the Client UI to uninstall applications.

The AIM manages application installs on the client machine. It keeps track of what applications have been installed on the client machines, where they have been installed and the various components that are part of the installation. It contacts the application servers to get the AppInstallBlock. The AIM uses the AppInstallBlock to then make the appropriate calls to the file spoofer; to install some files on the local disk; to "warm" the cache and to update the start menu and other short cuts as needed.

ECM

The ECM is part of the ECE. Its goal is to:

- Handle all file requests from the EFSD, either by using previously cached contents or requesting the contents from a server.

- Work with the LSM to insure that all applications have appropriately validated licenses before their files are accessed.

The ECM handles the volatile and non-volatile eStream cache on the client machine. It performs demand fetching from the appropriate server(s). Based on the client's observed behavior, it compiles updated profiling data, which may periodically be uploaded to a server.

EPF

The ECM is part of the ECE. Its goal is to intelligently use prefetching of file data to reduce latency of pages requested from the EFSD; this prefetching may result from profiling data or heuristics.

CNI

The client network component is the common point of connection between the rest of the eStream client components and the various eStream servers. Any client module that calls an interface of a server does so through the network component.

EMS

CBM

CIN

The client installer is a simple InstallShield (or simpler) application that will install all of the required client software.

FSP

The purpose of the file spoofer is to redirect file system accesses from some non-eStream drive. This may be necessary in order to support applications running under eStream that are hard-wired to access files in a specific location. The file spoofer may also be used if we are interested in providing a version of some system file different from the one actually on the client machine.

The file spoofer will intercept File Create calls for files that we are interested in spoofing and ensure that these creates are redirected to a file we specify. The redirection could be to a file on the EFSD, or to another, non-eStream'ed file.

EFSD

The EFSD provides standard kernel file system interfaces to the I/O manager and other kernel-mode components. It works with the NT Cache Manager to efficiently cache file

and directory contents. Its view of the ECM is essentially like that of a disk driver, sending primarily read and write requests as needed.

No Cluster

The VM clustering disabling driver (aka NoCluster) disables virtual memory clustering under Windows. While we don't fully understand all the implications, using this driver substantially reduces the average file system paging request size and can dramatically improve performance of eStream, especially on slower connections.

Virtual memory clustering, as implemented in Windows NT/2000, is intended to improve performance when paging to and from physical disks. If possible, we would like to disable clustering only for those threads/processes that will be doing a significant amount of I/O to the eStream file system.

Server components

The following are the server components for eStream 1.0:

- App Server. This is essentially a file server for eStream sets. It satisfies requests for pages from eStream files from the client.
- Slim Server. This handles requests from a client for user and service provider information, and grants access tokens to the client for executing eStream applications.
- Web Server. ???
- Monitor. This enables an administrator to view the server components. It regularly pings the various servers, takes disabled ones off line, and adds new ones to the pools.
- eStream Database. This tracks all user information and server resources for a given service provider.

App server

The application server is there to handle read requests for files accessed by eStream clients. Any file accessed on a client through the EFS can have this read request passed to an app server.

This will be the hardest working eStream server. It will respond to both synchronous (demand fetching) and asynchronous (prefetching) page requests from many different clients, for many different types of applications and files within those applications.

Slim server

The Software License Management (Slim) server is responsible for:

- Managing data related to users, the groups they belong to, and the applications they are subscribed to
- Validating the licenses for applications executing on clients
- Tracking all outstanding licenses currently in use

ASP web server

This describes only those interfaces on an ASP web server that relate to handling eStreamed applications.

Logically, the ASP web server is the backend web interface for user requests—e.g., get billing information, subscribe to a new app, or request a list of all possible apps a user can subscribe to. In the current model, the web server doesn't actually handle these requests, but instead passes them on to the appropriate eStream-centric server.

Monitor

The monitor utility is responsible for monitoring the overall health of the system. It is responsible to report server status, server traffic, illegal access etc. It will ping the Application Server and the Slim servers to gather the statistics and display them.

Database

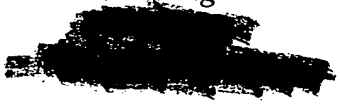
Builder components

These are the builder components for eStream 1.0:

- ???

eStream Web Server Load Monitoring Applet Low Level Design

Jae Jung



Functionality

One of the requirements for the eStream web server is a facility for monitoring server load (eStream requirements 3.0, 3.2, 3.4, 3.5). Per this document, this facility will be provided by a graphical load-monitoring applet that will be available for deployment at customer sites as part of the eStream web server installation.

The load-monitoring applet will present information in the following formats through a graphical interface:

- Real-time server load information plotted on strip chart
- Historical server load information plotted on line chart
- Multi-server real-time or historical load information on a line chart

Requirements

The following list details the provisional requirements for the load-monitoring applet. The remainder of this design document is based on these requirements.

1. The applet will be able to display server loads in "real-time" as load data is retrieved from the server.

2. The applet will be able to display (in a separate mode from real-time monitoring) historical load information to the extent that this information is available in the database.

3. The applet will be configurable (via applet parameters) for the following settings:
   -- Data retrieval rate, i.e., the frequency with which the applet request new load data from the web application server
   -- Chart window size, i.e., the number of datapoints shown in the chart window at any one time.

4. The applet will be capable of concurrently displaying the load of multiple servers in a clear and concise fashion.

5. The applet will support the displaying of cumulative and average load statistics for multiple servers.

6. In real-time mode, the applet will operate as a strip chart, with the fastest chart speed determined by a global configuration setting in database, typically corresponding to the frequency with which the eStream Monitor inserts load records into the database.

7. The applet will retrieve load information from the database via an http connection to the eStream web app server.

8. The applet will run in browsers with Java 1.0.x and 1.1.x support.

9. Internationalization support. Both the Applet and the backend pieces should be internationalizable.

Description

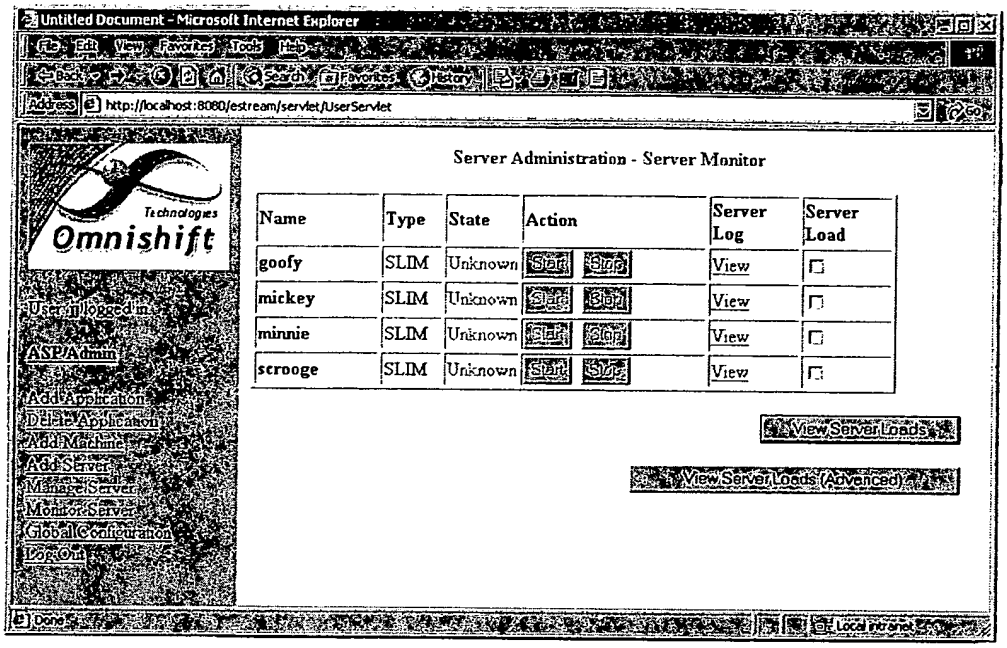
The load monitoring applet is comprised of two components. The first component will manage the retrieval of real-time or historical server load information from the eStream database. The second will consume this data and present this information to the user in a clear and concise graphical format. In addition to the applet, server-side objects will need to be written or extended to service data requests from the applet.

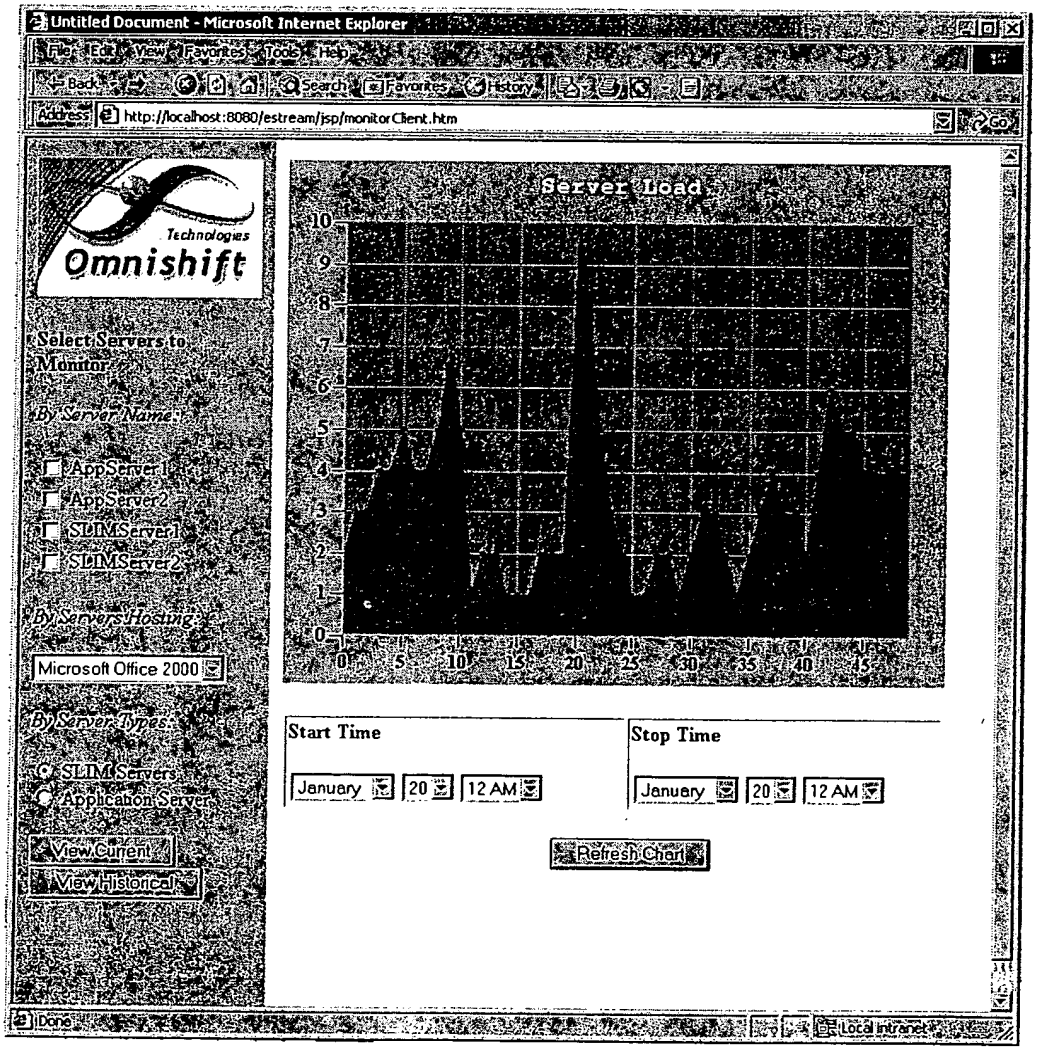
For the alpha-release of the eStream web server, the graphical presentation component will not be written internally; rather a commercially available applet or package will be used. Other aspects of the applet and the server-side components will be implemented to facilitate transitioning to an internally developed graphical component if such a decision is made for later releases.

User Interface Design

The following two screen shots are representative of the way that the load monitoring applet might be used in a particular monitoring/administration Brower interface. The first shot shows a general server administration and monitoring UI and the second shows a detailed load monitoring UI within which the load monitoring applet will be embedded.

The UI options shown in the second shot illustrate some of the reporting options for which the load monitor will be initially configurable. The applet(s) will be readily extensible to generate additional reports and more complex data combinations if desirable.





Interface Definitions

The load monitoring applet has two interfaces with other webserver components: <APPLET> tag parameters for its configuration within the web browser page and the HTTP request/response format with the web application server to request and receive load data.

1. **Applet <-> HTML Page Interface (<APPLET> Tag Parameters):**

Parameter	Req'd?	Significance	Default
hostName	Yes	Host name of webserver	None
hostPort	No	Host port of webserver	"80" (int)
chartSpeed	No	Time (in seconds) between chart scroll; also the resolution of the x-axis	"5" (int)
updatePeriod	No	Interval (in seconds) between HTTP requests for server load data. Note if chartSpeed < updatePeriod, the applet buffers load data for presentation. Maximum information latency will at most be equal to this value plus round trip time for the data request.	"10" (int)
chartWidth	No	Width (in ticks) of the applet chart; in conjunction with chartSpeed, implicitly determines the amount of data displayed	"50" (int)
mode	No	"current" for real-time monitoring and "history" for historical	"current"
startDate	No	if mode="history", the starting date for which the historical load chart will be plotted. Note if mode="current", this parameter is ignored.	none; value must be in Java Date format
stopDate	No	if mode="history", the stop date for which the historical load chart will be plotted. Note if mode="current", this parameter is ignored. Note that stopDate override the chartSpeed setting insofar as x-axis resolution is concerned.	none; value must be in Java Date format
numServers	Yes	number of servers to be plotted	1 (int, max 50)
serverName1 serverName2 ... serverName50	Yes	the id of the server(s) being plotted (up to 50 servers can be plotted at once). This **must** correspond to a existing serverID in the Server database table	none
serverData1 serverData2 ... serverData50	No	an initial set of data with which to display the initial chart. Note that these parameter(s) are the default means by which historical load data is displayed. Each set should be a comma delimited list of numbers (float)	none

- Note that applet tag parameters are all strings; where the applet does an internal type conversion, the target type format of the particular parameter has been noted.

- Note that the parameters determining the appearance of the chart (e.g., line colors, grid painting options, custom labels, etc.) are not included in this list. These parameters will either be determined by the parameters available for configuration in the commercial charting applet or TBD at some later date if the charting component is developed internally.

2. Applet <-> Web App Server Interface (HTTP)

Input:

The load monitoring applet will request load data from the webserver by executing the web server's MonitorServlet with the following parameters:

href="/MonitorServlet?action=getLoadData
&serverId=...
&startDate=...
&stopDate=...
&numPoints=...

where:

- serverId is a comma-delimited list of one or more server(s) for which load information is being requested. Note that this serverId corresponds to the serverID attribute in the Server database table.
- startDate is the (exclusive) starting date (in Java Date format) for which the load information is being requested
- stopDate is the (inclusive) end date (in Java Date format) for which the load information is being requested
- numPoints is the number of data points requested between the start and stop dates.

The applet will process the return data points as equally time-spaced points between the requested start and stop dates.

In addition, if startDate=stopDate, only one load data point (i.e., the most recent) will be returned by the servlet.

Output:

The load monitoring applet will expect load information from the web server via HTTP response in the following XML-like format:

<loadData>

```
<serverid=a >
        <LOADLIST>
                <LOAD value=x1/>
                <LOAD value=x2/>
                <LOAD value=x3/>
        ...
                <LOAD value=xN/>
        </LOADLIST>
</server>
<server id=b>
<LOADLIST>
                <LOAD value=y1/>
                <LOAD value=y2/>
                <LOAD value=y3/>
        ...
                <LOAD value=yN/>
        </LOADLIST>
</server>
        ...
</loadData>
```

In the above example, x1 to xN represents load values for the server a (by serverID), and y1 to yN represents load values for server b.

Testing Design

The load monitoring applet and related server-side Java code will need to be tested according to the plan outlined for other web server components in the Web Server/Database Low Level Design (WebServerDB-LLD.doc). Additionally, it should be noted that certain applet-related parameters (i.e., updatePeriod) that affect the frequency with which the applet requests load data from the web server are good candidates for tuning in order that a good balance between UI/measurement response and web server response performance be struck.

Upgrading/Supportability/Deployment Design

Upgrading/Supportability/Deployment Design will follow the model described for the Web Server in the Web Server/Database Low Level; Design (WebServerDB-LLD.doc).

Open Issues

1. How much will it cost to deploy the chosen commercial java applet/package(s) as an OEM installation? We need to find this out before we decide on a commercial

package. Also, another criteria for the choice would be: will we get support. Do we get the source code too?

2. Longer term, where's the cost/benefit breakpoint for the above where it makes more sense to write our own charting applet/package?